RHODES UNIVERSITY

November Examinations - 2010

Computer Science 301 - Paper 2 - Solutions

Examiners:	Time 3 hours
Prof P.D. Terry	Marks 180
Prof D.G. Kourie	Pages 15 (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".

Section A: Short questions [110 marks]

QUESTION A1 **[1 + 3 + 4 + 4 = 12 marks]**

(a) What is the essential difference between native-code compilation and interpretive compilation?

A native code compiler generates true machine code for the target processor, while an interpretive compiler generates code for an idealized machine, which has to be interpreted when the program is executed.

(b) In the practicals of this course you have used several compilers, in particular BCC and CL (C++ compilers), javac and jikes (Java compilers), FPC (Pascal), JPI (Modula-2), X2C, JAD, Parva and Coco. Which of these are interpretive compilers (+ 1 for each correct answer, -1 for each wrong answer)?

Parva is an obvious one. javac and jikes produce code for the JVM and so are also interpretive (of course, modern Java systems "JIT" this code at run time rather than interpret it slowly).

(c) Interpretive compilation suffers from the disadvantage that the object code runs more slowly than code produced by native code compilation. Why should this be, and what techniques are used by compiler writers to minimize the effect?

Effectively the code for the idealised machine has to go through a sort of translation process as the interpreter runs; interpreting each idealised opcode results in the execution of potentially many real machine instructions. The effects can be minimized by using sophisticated JIT techniques, or by developing the interpreter to be very efficient - for example by using clever assembly language and/or optimization compilers to develop it.

(d) If interpretive compilation suffers in this way, what makes it attractive to compiler writers? Suggest two situations where interpretive compilation can be put to very good use.

Interpretive systems are very useful for (a) prototyping the implementations of new languages (b) generating the initial compilers for existing languages on new architectures or operating systems and (c) as in the case of the JVM, allowing for highly portable object code - simply by implementing a new interpreter one can gain access to much precompiled code.

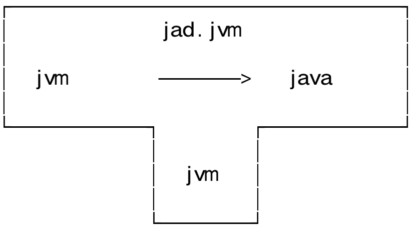
QUESTION A2 **[2 + 2 + 3 + 5 = 12 marks]**

In the practicals of this course you experimented with a decompiler named JAD.

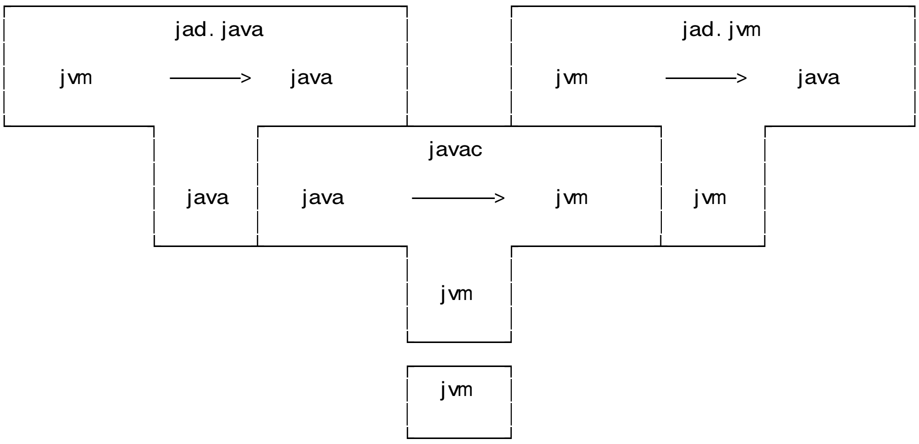
(a) What distinguishes a "compiler" from a "decompiler"?

A compiler typically translates from high-level source code to low-level object code. A decompiler does the *opposite - re-creates high level object code from low-level source code. Decompilers are much harder to implement, for fairly obvious reasons.*

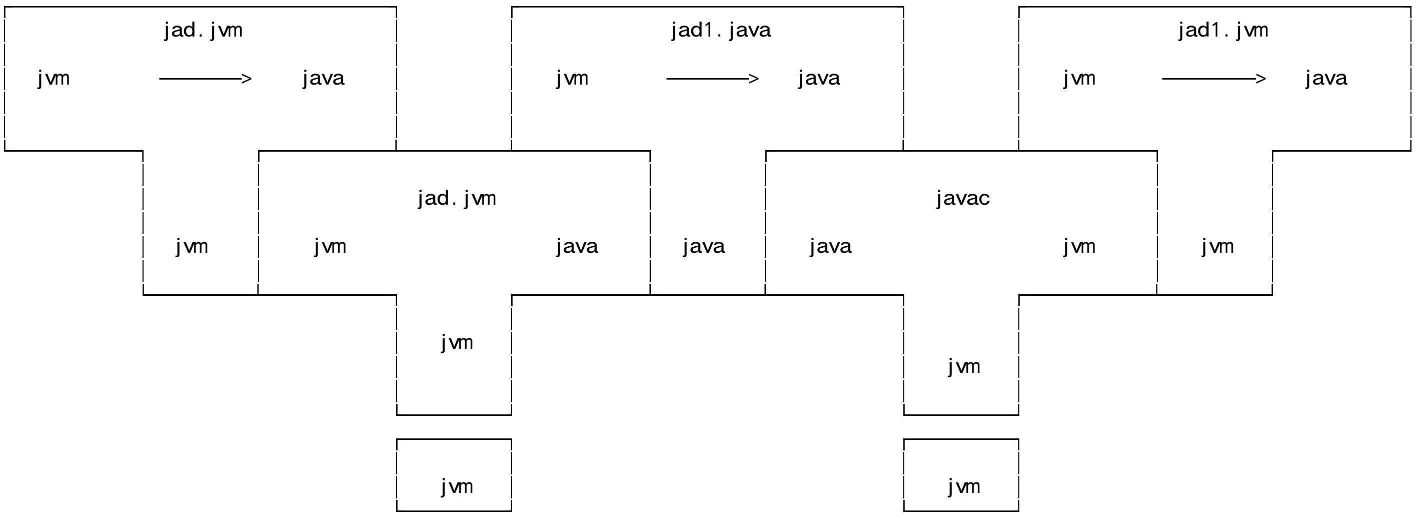
(b) Draw a T-diagram that captures the essence of the JAD decompiler.



(c) The developers of JAD may well have used Java in the development process. Given that they had a Java compiler like javac that could run on the JVM, draw T-diagrams illustrating how the final version of JAD (which you ran on the JVM) might have been produced.



(d) Using further T-diagrams, suggest and explain how the developers of JAD might have performed a self-consistency test of their product before releasing it to users.



Using jad to decompile "itself" should in principle return its own source code. In practice it might not be quite the same (comments omitted, layout slightly different, internal variable names different).

So denote the output of this stage by jad1.java.

However, compiling jad1.java *will generate* jad1.jvm, *which should be the same as* jad.jvm *again.*

QUESTION A3 **[1 + 8 + 2 + 1 = 12 marks]**

Scot MacHine has been set the task of developing a compiler for Canntaireachd. He wishes to allow the source code the possibility of incorporating comments of the sort familiar from the C family of languages:

```
/* comment text */
```

Since he planned to use Coco/R as a development tool he could simply have included the directive

```
COMMENTS FROM "/*" to "*/"
```

but his supervisor requested that he explore the use of the PRAGMAS facility instead. So Scot came up with the idea of writing

```
CHARACTERS
  inComment = ANY - "*" - "/" .
PRAGMAS
  Comment = "/*" { inComment } "*/" .
```

However, the regular expression forming part of this directive has some shortcomings.

(a) Here are some examples of strings that might be valid comments. One of them, however, cannot be a comment. Which is it?

```
(1)   /* comment */
(2)   /* multiply a * b */
(3)   /* divide a / b */
(4)   /* ***** */
(5)   /*********/
(6)   /* ** /* // */
(7)   /* // */ ** */
(8)   /*a*b*/
(9)   /* **a/b* ** */
```

The non-comment is (7) The first part of this is a comment / // */ and the rest is spurious text.*

(b) When it was pointed out to Scot that the text within a comment can validly include / and * characters, he came up with the refinement

```
CHARACTERS
  inComment = ANY - "*" - "/" .
PRAGMAS
  Comment = "/*" { inComment | "/" | "*" inComment } { "*" } "*/" .
```

Casual testing suggested to him that the problem was solved. However, this more complicated regular expression will still not handle some of the valid comments in (a). Identify two of these, and suggest how the regular expression might be improved still further.

It will still not recognize 4, 6 , 7, 9. A better definition takes some time to think through:

```
CHARACTERS
  inComment = ANY - "*" - "/" .
PRAGMAS
  Comment = "/*" { inComment | "/" | "*" { "*" } inComment } { "*" } "*/" .
```

(c) What practical advantage might there be in using the PRAGMAS facility of Coco/R rather than the COMMENTS facility when developing a compiler?

Pragmas, like comments, do not properly form part of source code. Both may often be ignored - but the ability to add an "action" to a pragma is often exploited in real compilers to set compilation options, as students must have realised from the practical course.

(d) In what applications is the Canntaireachd language useful?

It is a language like Tonic SolFa, but specifically for teaching classical Scottish bagpipe music (known as Piobaireachd).

QUESTION A4 **[8 + 6 + 4 = 18 marks]**

Formally, a grammar G is 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 formally we can define a language $L(G)$ produced by a grammar G by the relation

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

(a) In terms of this notation, express **precisely** (*that is to say, mathematically; we do not want a long essay or English description*) what you understand by

(a) A context-free grammar

A grammar is context-free if the left side of every production consists of a single non-terminal, and the right side consists of a non-empty sequence of terminals and non-terminals, so that productions have the form

$$\alpha \rightarrow \beta \qquad \text{with } | \alpha | \le | \beta | \ , \ \alpha \in N \ , \ \beta \in (N \cup T)^+$$

that is

$$A \rightarrow \beta \qquad \text{with } A \in N \ , \ \beta \in (N \cup T)^+$$

this is usually relaxed to allow nullable productions as well:

$$A \rightarrow \beta \qquad \text{with } A \in N \ , \ \beta \in (N \cup T)^*$$

(b) A sentential form

A sentential form is any string σ that is derivable from S. It does not have to be composed of terminals only (although it could be; in that case it has become a sentence).

$$\sigma \mid S \Rightarrow^* \sigma; \sigma \varepsilon (N \cup T)^*$$

(c) FIRST(A) where $A \in N$

The FIRST set is best defined by

$$a \in \text{FIRST}(A) \quad \text{if } A \Rightarrow^+ a\zeta \quad (A \in N \ ; \ a \ \varepsilon \ T \ ; \ \zeta \in (N \cup T)^* \ )$$

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

It is convenient to define the **terminal successors** *of a non-terminal A as the set of all terminals that can follow A in any sentential form, that is*

$$a \in \text{FOLLOW}(A) \quad \text{if } S \Rightarrow^* \xi A a \zeta \quad (A, S \in N \; ; \; a \in T \; ; \; \xi , \zeta \in (N \cup T)^* \;)$$

(b) What constraints must be imposed upon a context-free grammar if it is to be classified as an "LL(1) grammar"?

Rule 1

For each non-terminal $A_i \in N$ that admits alternatives

$$A_i \rightarrow \xi_{i1} \mid \xi_{i2} \mid \; . \; . \; . \; \xi_{in}$$

the sets of initial terminal symbols of all strings that can be generated from each of the alternative ξ_{ik} must be disjoint, that is

$$\text{FIRST}(\xi_{ij}) \cap \text{FIRST}(\xi_{ik}) = \emptyset \qquad \text{for all } j \neq k$$

Rule 2

For each non-terminal $A_i \in N$ that admits alternatives

$$A_i \rightarrow \xi_{i1} \mid \xi_{i2} \mid \; . \; . \; . \; \xi_{in}$$

but where $\xi_{ik} \Rightarrow \varepsilon$ for some k, the sets of initial terminal symbols of all sentences that can be generated from each of the ξ_{ij} for $j \neq k$ must be disjoint from the set FOLLOW(A_i) of symbols that may follow any sequence generated from A_i, that is

$$\text{FIRST}(\xi_{ij}) \cap \text{FOLLOW}(A_i) = \emptyset, \qquad j \neq k$$

or, rather more loosely,

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

(c) The syntax of many programming languages can be accurately described by a context-free grammar, yet most programming languages incorporate context-sensitive features. Give an example of a context-sensitive feature of the Parva language studied in this course. How does the Parva compiler handle this feature correctly?

There are plenty of examples to choose from - such as

- *variables must be declared before they are used*
- *the number of formal and actual arguments for functions must agree*
- *expressions used to control if and while statements must be of Boolean type*
- *values assigned to variables must be of the appropriate type*
- *break statements may only be used in the context of a loop or switch statement*

Many of these are usually handled by making use of a "symbol table" in which the various properties of the items denoted by identifiers are recorded, or by checking one count against another.

Notice that the "dangling else" phemomenon is not an example of a context sensitive feature. For some reason many candidates thought it was, but I cannot believe I ever said anything to promote this idea. It is, of course, a well known ambiguity, but ambiguity and context sensitivity are not the same thing!

QUESTION A5 **[12 + 8 + 2 + 16 + 2 + 4 = 44 marks]**

The contents page of a very useful textbook (if you can find a copy) begins as follows:

```
All you need to know to be able to pass your compiler examination.

                               by

                             Pat Terry.

Chapter 1  Bribery is unlikely to succeed.

Chapter 2  Understand the phases of compilation.
    2.1    Lexical and syntactic analysis are easily confused
    2.2    Constraint analysis involves the concept of type
    2.3    Code generation for the PVM is a breeze

Chapter 3  Get clued up on grammars.
    3.1    Terminals
    3.2    Sentences and sentential forms
    3.3    Productions
    3.4    EBNF and Cocol
    3.5    Ambiguity is bad news
```

The following Cocol grammar attempts to describe this contents page (and others like it - there may be many chapters and many subsections, of course, and some components are optional):

```
COMPILER Contents
/* Describe the contents pages of a book
   P.D. Terry, Rhodes University, 2010 */

CHARACTERS
  uLetter = "ABCDEFGHIJKLMNOPQRSTUVWXYZ" .
  lLetter = "abcdefghijklmnopqrstuvwxyz" .
  letter  = uLetter + lLetter .
  digit   = "0123456780" .

TOKENS
  word    = letter { letter } .
  number  = digit { digit } .
  section = digit { digit } "." digit { digit } .

IGNORE CHR(0) .. CHR(31)

PRODUCTIONS // Version 1
  Contents   = [ Title [ "by" Author [ Date ] ] ] { Chapter } .
  Title      = word { word } "." .
  Author     = word { word } "." .
  Date       = "(" number ")" .
  Chapter    = "Chapter" number Title { Subsection } .
  Subsection = section word { word } .
END Contents.
```

(a) Compute the FIRST and FOLLOW sets for each of the non-terminals of this grammar

This was mos easily done by using Coco/R with a $F pragma, which several candidates legitimatley did!

```
FIRST(Contents )  = { word "Chapter" }  FOLLOW(Contents ) = { EOF }

FIRST(Title) = { word }                 FOLLOW(Title) = { EOF section "by" "Chapter" }

FIRST(Author) = { word }                FOLLOW(Author) = { EOF "(" "Chapter" }

FIRST(Date) = { "(" }                   FOLLOW(Date) = { EOF "Chapter" }

FIRST(Chapter) = { "Chapter" }          FOLLOW(Chapter) = { EOF "Chapter" }

FIRST(Subsection) = { section }         FOLLOW(Subsection) = { EOF section "Chapter" }
```

Most of these are pretty obvious; the two that may catch you out are that "Chapter" is an element of FOLLOW(Chapter) and of FOLLOW(Subsection). This is because of the braces that appear round Chapter and Subsection in the productions.

(b) Is this an LL(1) grammar? Justify your answer, don't just guess!

Yes, it is an LL(1) grammar. To justify this we could either transform the grammar to eliminate the meta-brackets (tedious) or, more directly, look at the nullable components within the productions, as the grammar does not explicitly make us of alternation symbols.

There are nullable components of the form `{ word }` *in the productions for* `Title`, `Author` *and* `Subsection`. *In the first two of these the FIRST set for the nullable component contains word, while the FOLLOW set contains ".", so there is no problem with those. In the case of* `Subsection` *the FIRST set of the nullable component contains only* `word`, *while the FOLLOW set, as shown above, contains* `EOF`, `section` *and* "`Chapter`"*, so there is again no problem.*

The productions for `Chapter` *and* `Contents` *contain various nullable components. In the case of* `Chapter` *there is no problem, as the relevant FIRST and FOLLOW sets are {* `section` *} and {* `EOF`, "`Chapter`" *}.* `Contents` *is slightly more complicated, but by inspection one can again see that there will be no problems, for similar reasons.*

(c) The above grammar allows the *Contents* to be completely empty. Comment on the following attempt to change it so that (i) if the *Title* appears the list of chapters is optional but (ii) if the *Title* is absent there must be at least one *Chapter*.

```
PRODUCTIONS // Version 1 modified
  Contents   =   [ Title [ "by" Author [ Date ] ] ] Chapter { Chapter }
               | Title [ "by" Author [ Date ] ] { Chapter } .
```

This is a valid way of expressing the constraint, but it clearly renders the grammr non-LL(1) as both options for `Contents` *have* `word` *in their FIRST sets, thus contravening "Rule 1". A much better change would be*

```
PRODUCTIONS // Version 1 modified rather better
  Contents   =   Title [ "by" Author [ Date ] ] { Chapter } | Chapter { Chapter }
```

(d) How would you add actions to the *original grammar* (Version 1) above so as to develop a system that could tell you the title of the chapter with the greatest number of subsections and also issue a warning if the contents turns out to be completely empty?

A simple solution using a few static variables will suffice, similar to many examples seen in the practical course. A C# version follows; the Java one is essentially identical!

```
using Library;

COMPILER Contents $NCF
/* Describe the contents pages of a book
   P.D. Terry, Rhodes University, 2010 */

static string chapter, maxChapter = "";
static int sections, maxSections = 0;

CHARACTERS
  uLetter = "ABCDEFGHIJKLMNOPQRSTUVWXYZ" .
  lLetter = "abcdefghijklmnopqrstuvwxyz" .
  letter  = uLetter + lLetter .
  digit   = "0123456780" .

TOKENS
  word    = letter { letter } .
  number  = digit { digit } .
  section = digit { digit } "." digit { digit } .

IGNORE CHR(0) .. CHR(31)

PRODUCTIONS // Version 1
  Contents                (. bool hasTitle = false; .)
  = [ Title               (. hasTitle = true; .)
      [ "by" Author [ Date] ]
    ]
    { Chapter             (. if (sections > maxSections) {
                               maxSections = sections;
                               maxChapter = chapter;
                             } .)
    }                     (. IO.WriteLine("Chapter with most subsections: " + maxChapter);
                             if (!hasTitle && maxSections == 0)
                               Warning("Completely empty contents"); .) .

  Title
  = word                  (. chapter = token.val; .)
    { word                (. chapter = chapter + " " + token.val; .)
    } "." .
    Author
    = word { word } "." .

    Date
    = "(" number ")" .

    Chapter                 (. sections = 0; .)
    = "Chapter" number
       Title { Subsection   (. sections++; .)
       } .

    Subsection
    = section word { word } .

  END Contents.
```

(e) Here are two other possibilities for the set of productions for this system:

```
PRODUCTIONS // Version 2
  Contents   = [ Sentence [ "by" Sentence [ "(" number ")" ] ] ] { Chapter } .
  Sentence   = Words "." .
  Words      = word { word } .
  Chapter    = "Chapter" number Sentence { Subsection } .
  Subsection = section Words .
END Contents.

PRODUCTIONS // Version 3
  Contents   = [ Words "." [ "by" Words "." Date ] ] { Chapter } EOF .
  Chapter    = "Chapter" number Words "." { section Words } .
  Words      = word { word } .
  Date       = [ "(" number ")" ] .
END Contents.
```

If I were to claim that these grammars are "equivalent" I would be using the word "equivalent" in a special way. What is meant by the statement "two grammars are equivalent"?

Two grammars G1 and G2 are equivalent if L(G1) is identical to L(G2), that is to say, if the set of strings defined by the one grammar is identical to the set of strings defined by the second one. This does not demand that the sentences themselves have identical parse trees, however.

(f) Even though the sets of productions are equivalent, a developer might have reasons for preferring one set over the others. Which of these sets do you consider to be the "best", and why?

The first set has made far more use of non-terminals named to capture the semantic ideas that could be usful in identifying the various components found on a contents page. It does this at the expense of "repeating" the definition of Title to give the one for Author. However, if one were to develop a system further (as is done in (c)) it becomes useful to have these semantic "hooks" available, even though syntactically they may appear redundant.

QUESTION A6 **[12 marks]**

Consider the description of the contents of a book in Question A5, using the productions in version 1.

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

```
EOFSym, noSym, wordSym, numberSym, sectionSym, bySym, periodSym, chapterSym, lparenSym, rparenSym
```

How would you complete the parser routines below? There is no need to incorporate the refinement suggested in (c) of Question A5 - simply show the syntax analysis. A spaced copy of this system appears in the free information, which you are invited to complete and hand in with your answer book.

These sorts of parser are very easily written. There are, sadly, various traps that people fall into - such as inserting calls to `getSym()` *where they are not needed, or leaving them out in places where they are needed!*

```
static void Contents () {
// Contents = [ Title [ "by" Author [ Date ] ] ] { Chapter } .
  if (sym.kind = wordSym) {
    Title();
```

```
      if (sym.kind == bySym) {
        getSym();
        Author();
        if (sym.kind == lparenSym) Date();
      }
    }
    while (sym.kind == chapterSym) Chapter();
  }

  static void Title () {
  // Title = word { word } "." .
    accept(wordSym, "word expected");
    while (sym.kind == wordSym) getSym();
    accept(periodSym, " . expected");
  }

  static void Author () {
  // Author = word { word } "." .
    accept(wordSym, "name expected");
    while (sym.kind == wordSym) getSym();
    accept(periodSym, " . expected");
  }

  static void Date () {
  // Date = "(" number ")" .
    accept(lparenSym, "( expected");
    accept(numberSym, "year expected");
    accept(rparenSym, ") expected");
  }

  static void Chapter () {
  // Chapter = "Chapter" number Title { Subsection } .
    accept(chapterSym, "Chapter expected");
    accept(numberSym, "number expected");
    Title();
    while (sym.kind == sectionSym) Subsection();
  }

  static void Subsection () {
  // Subsection = section word { word } .
    accept(sectionSym, "section number expected");
    accept(wordSym, "word expected");
    while (sym.kind == wordSym) getSym();
  }
```

Section B: Enumeration types in Parva [70 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.

Yesterday you made history when you were invited to modify the Parva compiler to allow users to introduce "enumeration types".

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.

Since the basic system was built under great pressure, some subtle points were conveniently overlooked, and some features were omitted. So continue now to answer the following unseen questions that will test your ability to refine the system further. As a hint, many of these refinements only require you to add or modify a few lines of code as it is currently written. Your answers should be given by showing the actual code you need, not by giving a long-winded description in English!

As usual, a suite of simple test programs has been supplied for your use.

QUESTION B7 [6 marks]

The compiler only handled *if* statements without *else* clauses. That is very restrictive! Extend the system to allow *else* clauses.

This was unbelievably badly done. Most candidates seemed to have no idea of the necessity for an unconditional *branch at the end of the "then" section of code, needed only if there is an "else" clause present!*

```
        IfStatement<StackFrame frame>         (. Label falseLabel = new Label(!known);
                                                 Label outLabel = new Label(!known); .)
        =  "if" "(" Condition ")"             (. CodeGen.branchFalse(falseLabel); .)
           Statement<frame>
*          (   "else"                         (. CodeGen.branch(outLabel);
*                                                falseLabel.here(); .)
*              Statement<frame>               (. outLabel.here(); .)
*            | /* no else part */             (. falseLabel.here(); .)
*          ) .
```

QUESTION B8 **[6 marks]**

Allow for the definition of an enumerated type (optionally) to include the declaration of variables of that type, as in

```
            enum Villains { Tom, Dick, Harry } ringLeader, fallGuy = Dick;
```

We simply modify the *EnumDeclarations* production:

```
        EnumDeclarations<StackFrame frame>    (. String name;
                                                 Entry enumEntry = new Entry();
                                                 TypeRec type = new TypeRec();
                                                 int value = 0; .)
        = "enum" Ident<out name>              (. type.type = Table.getNextType(name);
                                                 enumEntry.name = name;
                                                 enumEntry.type = type.type;
                                                 enumEntry.kind = Entry.Enum;
                                                 Table.insert(enumEntry);
                                                 PVM.enumRefs.add(new EnumIndex()); .)
            "{"
              OneEnum<type.type, value++>
              { WEAK "," OneEnum<type.type, value++>
              }                               (. type.max = value - 1;
                                                 enumEntry.max = type.max; .)
**          "}" [ [ "[]"                      (. type.type++; .)
*                 ] VarList<frame, type>
*               ]
           WEAK ";" .
```

*Most candidates failed to notice that one might declare arrays of enumerated types in this was as well - see the line marked ** above*

QUESTION B9 **[4 marks]**

Allow enumeration values to be used as indices in array access.

This is handled by a very simple change to the *Designator* production:

```
        Designator<out DesType des>           (. String name;
                                                 int indexType; .)
        =  Ident<out name>                    (. Entry entry = Table.find(name);
                                                 if (!entry.declared)
                                                   SemError("undeclared identifier");
                                                 des = new DesType(entry);
                                                 if (entry.kind == Entry.Var) CodeGen.loadAddress(entry); .)
           [  "["                             (. if (isRef(des.type)) des.type--;
                                                 else SemError("unexpected subscript");
                                                 if (entry.kind != Entry.Var)
                                                   SemError("unexpected subscript");
                                                 CodeGen.dereference(); .)
*                 Expression<out indexType>   (. if (!isOrdinal(indexType))
                                                   SemError("invalid subscript type");
                                                 CodeGen.index(); .)
              "]"
           ] .
```

QUESTION B10 **[10 marks]**

The system as provided does no checking that the results of casting operations remain "in bounds". For example, code like

```
        enum Villains { Tom, Dick, Harry };
        int myAge = 65;
        Villains ringleader = cast(Villains, myAge);
```

should generate a run-time error. Why does it not do so, and how can you improve on this?

Hint: The PVM implementation you were supplied yesterday and today has some useful opcodes that have not previously been used.

It does not do so because no checks at all are imposed on the casting process. It is not difficult to add these, but we need to extend the `Entry` class to allow it to record the maximum value that a value of a given type may assume. This was, in fact, done silently in the code which was supplied before the examination at the "16 hour" point.

We need to be able to define the `max` field for variables of enumerated types - it is easy enough to define it for the `char` type as well (255), and to set it to a bogus value of -1 for `bool` and `int` variables.

To do so we introduce another small wrapper class:

```
    class TypeRec {
    // Objects of this class are associated with types with limited ranges of values
      public int type;
      public int max;
    } // end TypeRec
```

We then make a host of small changes to those parts of the system that deal with declarations (again, these had been made already, without comment or explanation, in the code which was provided).

Armed with this machinery we can now deal with the casting operation rather better. The relevant part of the *Primary* production is modified:

```
          | "cast" "("
              (    Ident<out name>          (. entry = Table.find(name);
                                               if (!entry.declared)
                                                 SemError("undeclared");
                                               if (entry.kind != Entry.Enum)
                                                 SemError("enumeration type name expected");
                                               newType = entry.type;
  *                                            max = entry.max; .)
  *             | "int"                     (. newType = Entry.intType; max = -1; .)
  *             | "char"                    (. newType = Entry.charType; max = 255; .)
              )
            "," Expression<out type>        (. if (!isOrdinal(type))
                                                 SemError("ordinal type needed");
                                               type = newType;
  *                                            CodeGen.cast(max); .)
            ")"
```

And use is made of the new `cast()` code generation routine - also silently provided. Students who had really studied the code would have had a distinct advantage over those who did not! Lest it be felt that this was totally unfair, the code supplied originally had imposed range checks on the `char` type, and in the practical course this had been emphasized. So it should have been an easy step to realise that something similar would be needed for enumerated types.

```
    public static void cast(int max) {
    // Generates code to check that TOS is within the range 0 <= tos <= max
      if (max >= 0) {
        emit(PVM.rng); emit(max);
      }
    }
```

Use is also made of the new opcode in the PVM for range checking, which is interpreted as follows (and which will, of course, cover the `char` type as well).

```
case PVM.rng:
  target = next();
  if (inBounds(cpu.sp))
    if (mem[cpu.sp] > target || mem[cpu.sp] < 0) ps = badVal;
  break;
```

QUESTION B11 [10 marks]

Similarly, code like

```
enum Villains { Tom, Dick, Harry };
Villains ringleader = Harry, scab = Tom;
ringLeader++;
--scab;
```

should generate an error. Why does it not do so, and how can you improve on this?

It does not do so because the simple increment and decrement opcodes impose no general checks, although, as before, students in the practical course had seen them applied to the `char` type. To achieve this we need to modify the *AssignOrDeclareStatement* parser:

```
    AssignOrDeclareStatement<StackFrame frame>
                                       (. int expType;
                                          DesType des;
                                          TypeRec type;
                                          boolean inc = true; .)
    = (  Designator<out des>
         (                             (. if (des.entry.kind != Entry.Var)
                                            SemError("invalid assignment");
                                          if (!des.canChange)
                                            SemError("may not alter this variable"); .)
             (   AssignOp
                 Expression<out expType>  (. if (!assignable(des.type, expType))
                                            SemError("incompatible types in assignment");
                                          CodeGen.assign(des.type); .)
               | ( "++" | "--"         (. inc = false; .)
                 )                     (. if (!isOrdinal(des.type))
                                            SemError("ordinal type needed");
*                                         CodeGen.incOrDec(inc, entry.max); .)
             )

           |                           (. if (des.entry.kind != Entry.Enum)
                                            SemError("enumeration name expected");
                                          type = new TypeRec();
                                          type.max = des.entry.max;
                                          type.type = des.entry.type; .)
              [ "[]"                   (. type.type++; .)
              ] VarList<frame, type>
         )
       | ( "++" | "--"                 (. inc = false; .)
         ) Designator<out des>         (. if (des.entry.kind != Entry.Var)
                                            SemError("variable designator required");
                                          if (!des.canChange)
                                            SemError("may not alter this variable");
                                          if (!isOrdinal(des.type))
                                            SemError("ordinal type needed");
*                                         CodeGen.incOrDec(inc, des.entry.max); .)
      ) WEAK ";"
    .
```

We also need to modify the code generator to be driven by an upper limit, rather than simply by the distinction between `int` and `char`:

```
*  public static void incOrDec(boolean inc, int max) {
   // Generates code to increment or decrement the value found at the address currently
   // stored at the top of the stack.
   // If necessary, apply range check
*    if (max >= 0) {
*      emit(inc ? PVM.ince : PVM.dece); emit(max);
*    }
     else emit(inc ? PVM.inc : PVM.dec);
   }
```

and use two more of the (silently unheralded) new opcodes in the long-suffering PVM:

```
case PVM.ince:          // ++ with range check
  adr = pop();
  target = next();
  if (inBounds(adr))
    if (mem[adr] < target && mem[adr] >= 0) mem[adr]++;
    else ps = badVal;
  break;

case PVM.dece:          // -- with range check
  adr = pop();
  target = next();
  if (inBounds(adr))
    if (mem[adr] <= target && mem[adr] > 0) mem[adr]--;
    else ps = badVal;
  break;
```

If you study the full solution you will see that the increment and decrement operations in the *ForStatement* have not incorporated range checking. This might appear to be a silly thing to do, but consider the effects of trying to write code like

```
enum Villains { Tom, Dick, Harry };
Villains fallGuy;
for (fallGuy = Tom; fallGuy <= Harry; fallGuy++) write(fallGuy);
```

With range checking, this loop would "crash" when `fallGuy` was incremented for the last time. You can't have it both ways. If you want to program in a C-like way you have to run your programs without proper error checks. All the more reason why one should prefer the Pascal-like *ForStatement* where the termination of the loop can be properly controlled.

Of course it's not only the *ForStatement* that would cause trouble. Consider

```
fallGuy = Tom;
while (fallGuy <= max(Villains) {
  write(fallGuy);
  ++fallGuy;
}
```

One cannot "fail" the condition in the *while* loop until `fallGuy` exceeds `max(Villains)` - but by then it is too late. If you want the safety of enumeration types you have to write messy code to bypass the checks if and when they become a nuisance. You might try, for example

```
fallGuy = Tom;
while (fallGuy <= max(Villains)) {
  write(fallGuy);
  if (fallGuy == max(Villains)) break;
  ++fallGuy;
}
```

QUESTION B12 **[6 marks]**

Allow for functions `max()` and `min()` in expressions, whose purpose is to allow you to write code like

```
enum Villains { Tom, Dick, Harry };
Villains fallGuy;
for fallGuy = min(Villains) to max(Villains) write(fallGuy);
```

This was not intended to be special to the *ForStatement*. We modify the *Primary* production to include the option:

```
| ( "max" | "min"               (. min = true; .)
  ) "(" Ident<out name>         (. entry = Table.find(name);
                                   if (!entry.declared)
                                     SemError("undeclared");
                                   if (entry.kind != Entry.Enum)
                                     SemError("enumeration type name expected");
                                   type = entry.type;
                                   CodeGen.loadConstant(min ? 0 : entry.max); .)
  ")"
```

QUESTION B13 **[24 marks]**

Casting in the form we have suggested is messy. Modify the system so that it can be done with code like

```
Villains fallGuy = Villains(2);   // Harry
int  i = int(fallGuy);           // 2
char c = char(65);               // A
```

Once again, ensure that casting operations check that the values remain "in bounds".

This was meant as the "biggie" that would sort out the top students from the rest. It calls for several changes, and considerable insight into language design and LL(1) limitations. The simpler changes needed are those to alter the options in the *Primary* production that deal with specific casting to `int` or `char` types:

```
| "(" Expression<out type> ")"
| "char" "("
     Expression<out type>       (. if (!isArith(type))
                                     SemError("invalid cast");
                                   else type = Entry.charType;
                                   CodeGen.castToChar(); .)
  ")"
| "int" "("
     Expression<out type>       (. if (!isOrdinal(type))
                                     SemError("invalid cast");
                                   else type = Entry.intType; .)
  ")"
```

The more awkward one is the option in *Primary* that deals with *Designators*, because there are now three possibilities for a *Primary* element to be introduced by an identifier - it could refer to a constant or a variable (as before) or could introduce a cast. In the latter case the identifier *must* be followed by an expression in parentheses; in the other cases this expression must be *absent*. The code for the revised option in *Primary* follows, which is worthy of close study (essentially we are using semantic information to resolve a potential LL(1) problem).

```
        |  Designator<out des>          (. type = des.type;
                                           switch (des.entry.kind) {
                                             case Entry.Var:
                                               CodeGen.dereference();
                                               break;
                                             case Entry.Con:
                                               CodeGen.loadConstant(des.entry.value);
                                               break;
*                                            case Entry.Enum:
*                                              break;
                                             default:
                                               SemError("wrong kind of identifier");
                                               break;
                                           } .)
*         (                             (. if (des.entry.kind != Entry.Enum)
*                                            SemError("invalid cast"); .)
*              "(" Expression<out type>  (. if (!isOrdinal(type))
*                                            SemError("invalid cast");
*                                          else type = des.entry.type;
*                                          CodeGen.cast(des.entry.max); .)
*              ")"
*           |                           (. if (des.entry.kind == Entry.Enum)
*                                            SemError("wrong kind of identifier"); .)
*         )
```

QUESTION B14 **[4 marks]**

It would surely be preferable to have a statement like

```
for fallGuy = Tom to Harry write(fallGuy);
```

produce the output `Tom Dick Harry` rather than `0 1 2`. This turns out to be harder than it looks. Why?

This is, indeed, harder than it looks, for the reason that the identifiers in the source code are normally eliminated from the object code; the symbol table plays its important role at compile-time and is then discarded - and in, any case, names come in and out of scope with some abandon.

Although this exercise was deemed too complicated for the students to tackle in the time they had available, it is, of course, possible to develop a system that enters the names of enumeration constants into the "string pool" of the PVM along with an index of references to these entries, and if you have come this far you may be interested in how it can be done. We introduce yet another small wrapper class

```
class EnumIndex {
  public ArrayList<Integer> index = new ArrayList<Integer>();
} // end EnumIndex
```

and arrange for the PVM to maintain a list of objects if this type, one corresponding to each enumeration type, and with its internal list of integers recording the string pool entries for the enumerated names:

```
   public static ArrayList<EnumIndex> enumRefs = new ArrayList<EnumIndex>();
```

The declaration of enumeration types requires modification

```
    EnumDeclarations<StackFrame frame>    (. String name;
                                             Entry enumEntry = new Entry();
                                             TypeRec type = new TypeRec();
                                             int value = 0; .)
    = "enum" Ident<out name>             (. type.type = Table.getNextType(name);
                                             enumEntry.name = name;
                                             enumEntry.type = type.type;
                                             enumEntry.kind = Entry.Enum;
                                             Table.insert(enumEntry);
*                                            PVM.enumRefs.add(new EnumIndex()); .)
        "{"
          OneEnum<type.type, value++>
          { WEAK "," OneEnum<type.type, value++>
          }                              (. type.max = value - 1;
                                             enumEntry.max = type.max; .)
        "}" [ [ "[]"                     (. type.type++; .)
              ] VarList<frame, type>
            ]
       WEAK ";" .

    OneEnum<int type, int value>         (. Entry constant = new Entry(); .)
*   = Ident<out constant.name>           (. int i = PVM.addString(constant.name);
*                                            PVM.enumRefs.get(PVM.enumRefs.size() - 1).index.add(i);
                                             constant.value= value;
                                             constant.type = type;
                                             constant.kind = Entry.Con;
                                             Table.insert(constant); .) .
```

Code generation for a *WriteElement* changes:

```
    WriteElement                         (. int expType, formType;
                                             boolean formatted = false;
                                             String str; .)
    =   StringConst<out str>             (. CodeGen.writeStr(str); .)
      | Expression<out expType>          (. if (!(isOrdinal(expType) || expType == Entry.boolType))
                                               SemError("cannot write this type"); .)
        [ ":"  Expression<out formType>  (. if (formType != Entry.intType)
                                               SemError("fieldwidth must be integral");
                                             formatted = true; .)
        ]                                (. switch (expType) {
                                               case Entry.intType:
                                               case Entry.boolType:
                                               case Entry.charType:
                                                 CodeGen.write(expType, formatted); break;
*                                              default:
*                                                CodeGen.writeEnum((expType - Entry.enumType) / 2,
*                                                                  formatted); break;
                                             } .) .
```

with a new code generating routine

```
    public static void writeEnum(int type, boolean formatted) {
    // Generates code to output value of specified type from top of stack
      emit(formatted ? PVM.prnfe : PVM.prne);
      emit(type);
    }
```

and new opcodes, interpreted as follows:

```
case PVM.prne:          // enumerated string output from string pool
  if (tracing) results.write(padding);
  results.write(strPool.get(enumRefs.get(next()).index.get(pop())));
  if (tracing) results.writeLine();
  break;
case PVM.prnfe:         // enumerated string output from string pool formatted
  if (tracing) results.write(padding);
  fieldWidth = pop();
  results.write(strPool.get(enumRefs.get(next()).index.get(pop())), fieldWidth);
  if (tracing) results.writeLine();
  break;
```

Note that the call to `CodeGen.writeEnum()` might produce a meaningless argument, but since the code would not be executed, this does not really matter!

END OF EXAMINATION QUESTIONS

Section C

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

A complete system incorporating the features they had been asked to implement 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 Section B would probe this understanding. No 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. The system as supplied at this point was deliberately naive in some respects, in order to lay the ground for the unseen questions of the next day.

Section D

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

Candidates were provided with an exam kit for Java or C#, containing files for developing a Parva compiler like that which they had used in the practical course.

They were also given a description of enumeration types, and a suite of simple, suggestive test programs and asked to extend the compiler to handle these. Finally, they were told that later in the day some further ideas and hints would be provided.