RHODES UNIVERSITY

November Examinations - 2015

Computer Science 301 - Paper 2 (Solutions)

Examiners:
Prof P.D. Terry
Prof P. Blignaut

Time 4 hours
Marks 180
Pages 14 here

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

(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 develop graphics facilities for Parva. Some 16 hours before the examination a complete grammar and other support files for building such a 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 system, access to a computer, and machine readable copies of the questions.)

Section A [100 marks]

Section A: Conventional questions [100 marks]

QUESTION A1 **[5 marks]**

A1 *Pragmas* and *comments* often appear in source code submitted to compilers. What property do pragmas and comments have in common, and what is the semantic difference between them? [5 marks]

Solution:

Neither pragmas nor comments affect the dynamic semantics of a program, and can normally be ignored in that respect. They can usually appear in source anywhere that white space can be, that is, between any two tokens. But pragmas can choose compiler options, as students should have realized from using them in the practical course.

QUESTION A2 **[6 marks]**

A2 What do you understand by the term *short circuit semantics* as applied to the evaluation of Boolean expressions? Illustrate your answer by means of some specimens of code that incorporate simple Boolean expressions, distinguishing between short-circuit semantics and some other alternative. [6 marks]

Solution:

Standard bookwork sort of discussion expected, and students should know about this from first year in any case. An example might be

```
WHILE  (1 < P)  AND  (P < 9)  DO   P := P + Q  END

L0      if 1 < P goto L1   ; Short circuit version
        goto L3
L1      if P < 9 goto L2
        goto L3
L2      P := P + Q
        goto L0
L3      continue


L0      T1 := 1 < P        ; standard Boolean operator approach
        T2 := P < 9
        if T1 and T2 goto L1
        goto L2
L1      P := P + Q
        goto L0
L2      continue
```

Alternatively it is more or less adequate to point out that in short-circuit semantics

```
A and B   is equivalent to    if not A then false else B
A or  B   is equivalent to    if A then true else B
```

Essentially the important point is that it may not be necessary to evaluate both operands. This is useful in writing protective code like

```
if (p != null  &&  p.flag) ...
if (d != 0  &&  n / d > 12)  ...
```

QUESTION A3 **[12 marks]**

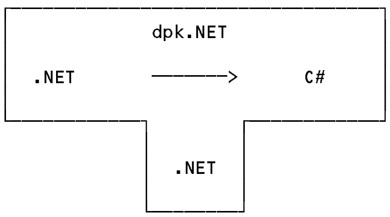
A3 In the practicals of this course you experimented with a decompiler named dotPeek.

(a) What distinguishes a "compiler" from a "decompiler"? [2 marks]

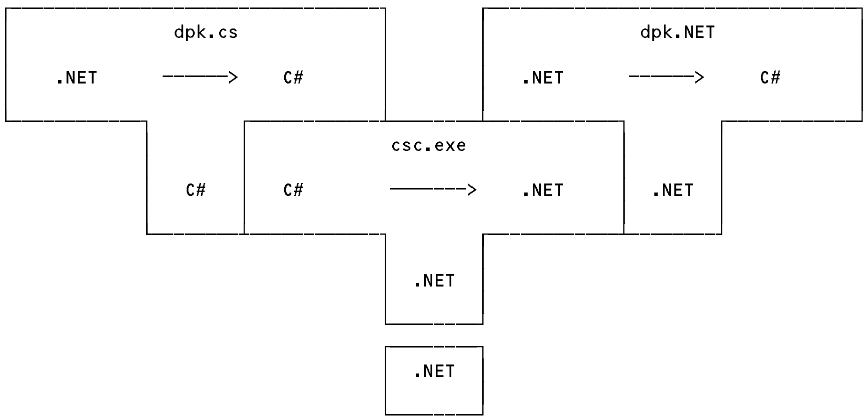
Solution:

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.

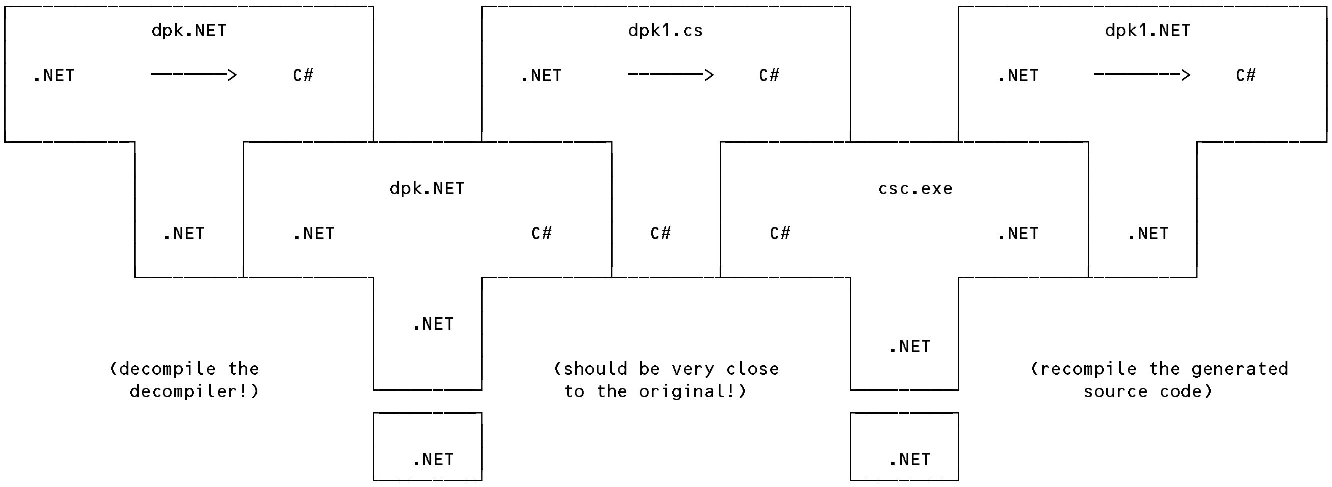
(b) Draw a T-diagram that captures the essence of the dotPeek decompiler. [2 marks]



(c) The developers of dotPeek may well have used C# in the development process. Given that they had a C# compiler like `csc.exe` that could run on .NET, draw T-diagrams illustrating how the final version of dotPeek (which you ran on .NET) might have been produced. [3 marks]



(d) Using further T-diagrams, suggest and explain how the developers of dotPeek might have performed a self-consistency test of their product before releasing it to users. [5 marks]



Using dpk.NET 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 `dpk1.cs`.

However, compiling `dpk1.cs` will generate `dpk1.NET`, which should be the same as `dpk.NET` again.

QUESTION A4 [15 marks]

A4 Consider the following Cocol specification:

```
COMPILER Expressions

CHARACTERS
  digit  = "0123456789" .
  letter = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .

TOKENS
  ident  = letter { letter | digit } .

IGNORE CHR(0) .. CHR(31)

PRODUCTIONS
  Expressions  = { Term "?" } EOF .
  Term         = Factor { "+" Factor | "-" Factor } .
  Factor       = ident [ "++" | "--" ] | [ "abs" ] "(" Term ")" .
END Declarations.
```

Develop a handcrafted scanner (not a parser!) that will recognize the tokens used in this grammar. Assume that the first few lines of the scanner routine are introduced as

```
static int getSym() {
  while (ch != -1 && ch <= ' ') getChar();
```

and that the `getSym()` method must return an integer representing the token that it has recognized, as one of the values that you can denote by the names:

```
plus, minus, plusPlus, minusMinus, query, lParen, rParen, abs, ident, error, EOF
```

Assume that when the scanner invokes the source handling method

```
static void getChar()
```

this will assign to a static field `ch` (of the class of which these methods are members) the next available character in the source text, or -1 if no more characters can be read. [15 marks]

Solution:

A method might go something like the one below (students had constructed scanners like this in practicals, though not this particular one). Typical errors are likely to be omission of `getChar()` calls - or writing a `void` method when a value returning one was required.

```
static int GetSym() {
  while (ch <> -1 && ch <= ' ') GetChar();
  if (ch = -1) return EOF;
  if (Char.IsLetter(ch)) { // identifier or keyword
    StringBuilder sb = new StringBuilder();
    while (Char.IsLetterOrDigit(ch) {
      sb.Append(ch);
      GetChar();
    }
    if (sb.ToString().Equals("abs")) return abs; else return ident;
  }
  switch (ch) {
    case '?': GetChar(); return query;
    case '(': GetChar(); return lparen;
    case ')': GetChar(); return rparen;
    case '+': GetChar();
              if (ch != '+') return plus;
              GetChar(); return plusPlus;
    case '-': GetChar();
              if (ch != '-') return minus;
              GetChar(); return minusMinus;
    default : GetChar(); return error;
  }
}
```

QUESTION A5 **[42 marks]**

A5 Consider the following grammar, expressed in EBNF, that describes the form of a typical university course:

```
Course       = Introduction Section { Section } Conclusion .
Introduction = "lecture" [ "handout" ] .
Section      = { "lecture" | "prac" "test" | "tutorial" | "handout" } "test" .
Conclusion   = [ "panic" ] "Examination" .
```

(a) What do you understand by the statement "two grammars are equivalent"? [2 marks]

Two grammars are equivalent if they can generate (describe) exactly the same set of sentences (not necessarily yielding the same parse trees or using the same sets of productions).

(b) What do you understand by the statement "a grammar is ambiguous"? [2 marks]

A grammar is ambiguous if there is at least one sentence that can be derived from the goal symbol in more than one way. This is a very simple definition, and it is alarming that students cannot explain ambiguity succinctly.

(c) Rewrite these productions so as to produce an equivalent grammar in which no use is made of the EBNF meta-brackets { ... } or [...]. [8 marks]

```
Course             = Introduction Section Sections Conclusion .
Sections           = Section Sections | ε .
Introduction       = "lecture" OptionalHandout .
OptionalHandOut    = "handout" | ε .
Section            = Components "test" .
Components         = ( "lecture" | "prac" "test" | "tutorial" | "handout" ) Components | ε .
Conclusion         = UnderstandablePanic "Examination" .
UnderstandablePanic = "panic" | ε .
```

(d) Clearly explain why the derived set of productions does not obey the LL(1) constraints. [4 marks]

OptionalHandout is nullable, and *FIRST(OptionalHandOut) = { "handout" }* while *FOLLOW(OptionalHandout)* is the set *FIRST(Section) = {"lecture", "prac", "tutorial", "handout"}*, and these two sets have *"handout"* in common.

(e) It is suspected that the original grammar might be ambiguous. Give an example of a sentence that would confirm this suspicion. [4 marks]

We only need to find one sentence that can be parsed in two ways. A fairly obvious example is

```
lecture handout test examination
```

can be parsed with `handout` regarded as either a component of the *Introduction* or as a component of the first *Section* after an *Introduction* that has no `handout`.

(f) Why does it not follow that every grammar that is not LL(1) must be ambiguous? Justify your argument by giving a simple example of a non-LL(1) grammar that is not ambiguous. [4 marks]

No, it does not follow - it is unidirectional. Ambiguous implies non-LL(1) but non-LL(1) does not necessarily imply ambiguous! The obvious example to cite here is a left recursive grammar for expressions: Left recursive grammars are always non-LL(1), but do not have to be ambiguous! A simple example would be

```
Expression = Expression ( "+" | "-" ) Term   |   Term .
Term       = "a" | "b" .
```

(g) Does it follow that if a recursive descent parser is constructed for this grammar, it is doomed to fail? Justify your answer. [3 marks]

No it does not. This is the famous "dangling else" in disguise. It would of course matter if there was a different semantic "value" for a `handout` that was issued as part of an *Introduction* or as part of a *Section* (which of course there might be - remember that the grammar as it stands just handles syntax). The sentence given earlier would be parsed by a recursive descent parser quite happily, binding the `handout` to the introductory `lecture`.

(h) The University Senate have decreed that courses may no longer be offered if they do not obey various constraints. In particular, an acceptable course may not

(1) offer fewer than 50 lectures, or subject the candidates to more than 8 tests;
(2) hold a practical until at least 4 lectures have been given;
(3) hold more practicals than tutorials.

Show how the grammar may be augmented to impose these constraints. A spaced copy of the grammar appears in the free information, which you are invited to complete and hand in with your answer book. [15 marks]

A simple solution follows. The system is simple enough that the totals needed can be held in static fields of the parser - any attempt to use parameters is rather overkill. The question was capable of various interpretation as it happened, and it would have been acceptable to report on an excessive number of tests at the end, rather than within a section.

These are rather trite questions - one does not really need a system like Coco/R to develop simple counting applications (although, as we can see, it makes it very easy to do so). This sort of question is useful for probing whether students can appreciate where attributes must be added to the grammar; beginners tend to insert them almost at random, but their placement is usually critical.

```
COMPILER Course

  static int lectures = 0, tutorials = 0, tests = 0, practicals = 0;

IGNORE CHR(0) .. CHR(31)

PRODUCTIONS
  Course
  = Introduction
    Section
    { Section
    } Conclusion               (. if (lectures < 50 || practicals > tutorials)
                                      SemError("Constraints on course not met") .) .

  Introduction
  = "lecture"                 (. lectures++ .)
    [ "handout" ] .


  Section
  = {   "lecture"             (. lectures++; .)
      | "prac"                (. practicals++;
                                 if (lectures < 4)
                                   SemError("Not enough lectures before this prac"); .)
        "test"                (. tests++;
                                 if (tests > 8)
                                   SemError("Too many tests");
      | "tutorial"            (. tutorials++; .)
      | "handout"
    }
    "test"                    (. tests++;
                                 if (tests > 8)
                                   SemError("Too many tests"); .) .

  Conclusion
  = [ "panic" ]
    "Examination" .

END .
```

QUESTION A6 [20 marks]

A6 Generations of Parva programmers have complained about the absence of a *for* loop, and it has been decided to add this feature, using syntax suggested by Pascal, and exemplified by

```
for i = 1 to 10 write(i);

for j = i + 1 to i - 4 {
  read(i); total = total + i;
}
```

(a) Write an EBNF description that should allow you to recognize such a statement, taking care not to be over-restrictive. [2 marks]

Solution:

ForStatement = Designator "=" Expression "to" Expression Statement .

The unwarranted restriction might arise if one limited *Designator* to `identifier` (say) or used `number` instead of *Expression*. Actually, using `identifier` would be useful if one wished to deal with (e) below more restrictively. There is no need to use *Block* in place of *Statement* - a *Block* is an acceptable form of *Statement*!

Some candidates might suggest

ForStatement = Assignment "to" Expression Statement .

but this is not a good idea. In Parva an *Assignment* is terminated by a semicolon, just for a start.

(b) What static semantic constraints must be satisfied by the various components of your production(s)? You might illustrate this by writing the Cocol attributes; alternatively just specify them in concise English. [4 marks]

Solution:

The two *Expressions* must yield values of a type that is at least assignment compatible with the type of the *Designator*, and these must be ordinal types (that is, map to simple integers). So in the context of the compiler students had worked with, one could have used an integer control variable with expressions of integer or character type, or a character control variable with expressions of character type, but not control variables and expressions of Boolean type or a control variable of character type and expressions of integer type. It would be too much to hope that candidates would see all those points, especially those who have been corrupted by coding in a near typeless language like C.

(c) Critically examine and comment in detail on the suggestion that a *for* loop of the form

```
for i = start to stop {
  // something
}
```

should be regarded as (dynamically) semantically exactly equivalent to the following: [8 marks]

```
i = start;
while (i <= stop) {
  // something
  i++;
}
```

Solution:

Aha - a language lawyer question! The breed known as the language lawyer was mentioned in class discussions - the sort of person that writes down silly example code and challenges the reader to say what it really means. Simple as they seem to be, *for* loops abound with possibilities for constructing funny examples that could get one into trouble.

There are two aspects of the *for* loop that should be emphasized that suggest it could be rather different from a simple *while* loop as suggested above. Firstly, in the *for* loop the system usually arranges to evaluate the limits imposed by the two *Expressions* once only, before the loop can begin. Secondly, it may be desirable to regard the *for* loop as creating another level of scope, with the scope if the control variable confined to the statement as a whole. To my delight (and, I must say, astonishment), when I set this problem some years ago, several students raised the scope issue (which I do not recall mentioning in class for a *for* statement; in fact I don't really remember disussing the *for* statement at all!)

To show up these problems, consider

```
for i = i to i + 4  {  // in principle never terminates if you interpret it as above
  // something
}

for i = i to maxInt {  // in practice never terminates (overflows and becomes negative)
  // something
}

for i = i to maxInt {  // in practice might never terminate - something messes with i
  // something
  i = 0;
}
```

Although it was not required in a solution, for enlightenment it may be of interest to show how these issues can be overcome by a fairly dramatic process. We arrange that the code generation for the generic loops (the *downto* loop is a pretty obvious variation and extension)

```
for Variable = Expression1 to Expression2 Statement
for Variable = Expression1 downto Expression2 Statement
```

must yield lower level code of the form

```
       Temp1 := Expression1                        Temp1 := Expression1
       Temp2 := Expression2                        Temp2 := Expression2
       IF Temp1 > Temp2 THEN GOTO EXIT             IF Temp1 < Temp2 THEN GOTO EXIT
       Variable := Temp1;                          Variable := Temp1;
BODY:  Statement                            BODY:  Statement
       IF Variable = Temp2 THEN GOTO EXIT          IF Variable = Temp2 THEN GOTO EXIT
       Variable := Variable + 1                    Variable := Variable - 1
       GOTO BODY                                   GOTO BODY
EXIT:                                       EXIT:
```

respectively, where `Temp1` and `Temp2` are temporary variables. This code will not assign a value to the control variable at all if the loop body is not executed, and will leave the control variable with the "obvious" final value if the loop body is executed. It may appear to be a little awkward for an incremental compiler, since there are now multiple apparent references to extra variables and to the control variable. However, it can be done for Parva and the PVM if one resorts to the usual trick of defining a few more opcodes.

(d) If a *for* loop is implemented as in (c) - or, indeed, in any other way - problems would arise if the code indicated by "something" were to alter the value of the control variable. Do you suppose a compiler could forbid this practice? If so, how - and if not, why not? [6 marks]

Solution: As we discovered when trying to standardize Modula, it is almost impossible to be completely watertight. However, if the symbol table entry for the control variable is marked in some way as "cannot be changed for the duration of the loop body" and if the productions for input statements, assignment statements and statements like `i++;` check that their "target" is not marked in this way, one can certainly handle the situation in simple Parva programs like those that students compiled, which had only a single main method. It gets more difficult if one can pass control variables as reference parameters, or if global variables (static fields) are permitted as control variables, and even more awkward if one can find other aliased pointers to control variables and attack them in that way! But one hopes that students will at least have thought of the idea of marking a symbol table entry as "don't touch" for the duration of parsing the loop body.

Section B [80 marks] Parva with simple graphics.

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 EXAMC.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 develop an extended Parva compiler which would provide support for simple graphics. 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. Time to ask a few searching questions!

QUESTION B7 **[4 marks]**

B7 The $ST, $SD and $HD pragmas are very useful, but a bit irritating to a simple user. How can the system be modified so that they only take effect if the $D debugging pragma has been turned "on"? [4marks]

This is simply achieved by modifying the PRAGMAS section of the grammar as shown

```
  **   StackDump   = "$SD" .           (. if (debug) CodeGen.Stack(); .)
  **   HeapDump    = "$HD" .           (. if (debug) CodeGen.Heap(); .)
  **   TableDump   = "$ST" .           (. if (debug) {
                                            Table.PrintTable(OutFile.StdOut);
                                            Types.Show(OutFile.StdOut);
                                          } .)
```

QUESTION B8 **[4 marks]**

B8 The syntax of the main `Parva` production is essentially:

Parva = { FuncDeclaration } EOF .

Is any practical advantage gained by the addition of the explicit EOF? Would the compiler not perform as well if EOF were to be omitted? Discuss briefly. [4 marks]

If the EOF is omitted there is a chance that a program with spurious rubbish after a function will appear to compile correctly. Consider a silly example like

```
void Main() {
}

voiddd Wrong() {
}
```

`voiddd` cannot start a function declaration, so the compilation will stop at that point without reporting that something has gone amiss.

QUESTION B9 [10 marks]

B9 "Anyone can write a compiler for error-free input". The system provided to you was clearly written in a hurry, and several checks have simply been omitted in the parts dealing with parameters for the Turtle Graphics statements. Identify some of these and suggest what semantic checking should be added, and where this must occur. [10 marks]

Well, once you have seen one, you have almost seen them all. Each call to *Expression* needs to have a type check. There are neater ways (aren't there?) but under exam conditions it is adequate to say:

```
       InitGraphicsStatement                (. int expType; .)
       =  "InitGraphics"
   **     "("      Expression<out expType>   (. if (!IsArith(expType))
   **                                             SemError("arithmetic argument needed"); .)
   **       WEAK "," Expression<out expType> (. if (!IsArith(expType))
   **                                             SemError("arithmetic argument needed");
                                               CodeGen.InitGraphics(); .)
          ")" WEAK ";"
       .

       MoveStatement                        (. int expType; .)
       =  "Forward"
   **     "("      Expression<out expType>   (. if (!IsArith(expType))
   **                                             SemError("arithmetic argument needed");
                                               CodeGen.Forward(); .)
          ")" WEAK ";"
       .

       TurnStatement                        (. int expType; .)
       =  "TurnLeft"
   **     "("      Expression<out expType>   (. if (!IsArith(expType))
   **                                             SemError("arithmetic argument needed"); .)
                                               CodeGen.TurnLeft(); .)
          ")" WEAK ";"
       .

       LineStatement                        (. int expType; .)
       =  "Line"
   **     "("      Expression<out expType>   (. if (!IsArith(expType))
   **                                             SemError("arithmetic argument needed"); .)
   **     WEAK "," Expression<out expType>   (. if (!IsArith(expType))
   **                                             SemError("arithmetic argument needed"); .)
   **     WEAK "," Expression<out expType>   (. if (!IsArith(expType))
   **                                             SemError("arithmetic argument needed"); .)
   **     WEAK "," Expression<out expType>   (. if (!IsArith(expType))
   **                                             SemError("arithmetic argument needed");
                                               CodeGen.Line(); .)
          ")" WEAK ";"
       .
```

Not quite so obvious perhaps is that the `Length()` function (a component of *Primary*) needs some checking:

```
   **   | "Length" "(" Designator<out des> (. if (des.entry.kind != Kinds.Var)
   **                                          SemError("variable expected");
   **                                        if (!IsArray(des.type))
   **                                          SemError("not an array");
                                             CodeGen.Dereference();
                                             CodeGen.Dereference();
                                             type = Types.intType; .)
```

QUESTION B10 **[8 marks]**

B10 The Turtle Graphics commands have been restricted to `Forward` and `TurnLeft`. Can the Parva system be modified to allow obvious extensions `Backward` and `TurnRight`, without adding to the PVM and code generator classes? If so, how, and if not, why not? [8 marks]

This is easily achieved by arranging that the code for `NegateInteger` is generated ahead of the code for moving or turning the turtle, if `Backwards` or `TurnRight` are encountered:

```
    MoveStatement                        (. int expType;
 **                                         bool back = false; .)
 ** =  (   "Forward"
 **      | "Backward"                    (. back = true; .)
 **    )
 **    "("      Expression<out expType>  (. if (!IsArith(expType))
 **                                           SemError("arithmetic argument needed"); .)
 **    ")"                               (. if (back) CodeGen.NegateInteger();
                                            CodeGen.Forward(); .)
       WEAK ";"
    .

    TurnStatement                        (. int expType;
 **                                         bool right = false; .)
 ** =  (   "TurnLeft"
 **      | "TurnRight"                   (. right = true; .)
 **    )
 **    "("      Expression<out expType>  (. if (!IsArith(expType))
 **                                           SemError("arithmetic argument needed"); .)
 **    ")"                               (. if (right) CodeGen.NegateInteger();
                                            CodeGen.TurnLeft(); .)
       WEAK ";"
    .
```

QUESTION B11 **[6 marks]**

B11 Two students were overheard comparing possible ways of implementing the *if-then-else* construct in Parva. One of them had developed the solution suggested in the kit:

```
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(); .)
 )
 .
```

while the other had come up with something slightly different:

```
IfStatement<StackFrame frame>     (. Label falseLabel = new Label(!known);
                                     Label outLabel = new Label(!known); .)
=  "if" "(" Condition ")"         (. CodeGen.BranchFalse(falseLabel); .)
      Statement<frame>            (. CodeGen.Branch(outLabel);
                                     falseLabel.Here(); .)
   [ "else" Statement<frame> ]    (. outLabel.Here(); .)
.
```

Which of these "works"? Perhaps both "work"? In that case, which, if either, is better than the other, and why? [6 marks]

They both "work", but the first generates better code than the second in the situation where the "else" clause is absent. The second version generates an unnecessary BRN instruction (it causes no real harm, of course).

```
    if (test) b; else c;              first case        if (test) b;      second case

      code for test                   code for test                       code for test
      BZE  L2                         BZE  L2                             BZE  L2
      code for b;                     code for b;                         code for b;
      BRN  L3                     L2  continue                            BRN  L2       *** unnecessary
 L2   code for c;                                                     L2  continue
 L3   continue
```

QUESTION B12 **[24 marks]**

B12 Here are two trivial programs that will currently compile without complaining but must surely misbehave. How should the system be improved to handle this? Can problems like this be anticipated and detected at compile time? If so, how? If not, could an improved implementation of the PVM help? Show how this could be achieved. [24 marks]

```
void Main() {
  DrawLine(0, 0, 50, 50);
} // Main

void Main() {
  InitGraphics(500, 500);
  DrawLine(0, 0, 50, 50);
  CloseGraphics();
  DrawLine(50, 50, 0, 100);
} // Main
```

The problem, hopefully obvious, is that if the call to `InitGraphics` does not precede calls to `Forward`, `TurnLeft` etc then the canvas will not have been instantiated and thus an exception will be raised, which is anything but user friendly. This cannot in general be detected at compile-time using the simple techniques covered in this course, as the various paths through the methods and statements are only known at run-time.

The solution is simply to keep track in the PVM of whether the `InitGraphics` code has been executed, and to complain if any of the other graphics opcodes can detect that this has not been done. Here is one way of doing it:

```
    static IntSet graphOps = new IntSet(homet, turnt, movet, penu, pend, line);  // graphics opcodes


      ps = running;               // prepare to execute
  **  bool graphicsInitialised = false;

      do {
        pcNow = cpu.pc;           // retain for tracing/postmortem
        if (cpu.pc < 0 || cpu.pc >= codeLen) {
          ps = badAdr;
          break;
        }
        cpu.ir = Next();          // fetch
        if (tracing) Trace(results, pcNow);

  **    if (!graphicsInitialised && graphOps.Contains(cpu.ir)) {
  **      ps = badGraph;
  **      break;
  **    }

        switch (cpu.ir) {         // execute
          case PVM.nop:           // no operation
            break;

          // ........... lots more

          case PVM.initg:         // initialize graphics window
            tos = Pop();
            sos = Pop();
            Turtle.InitGraphics(sos, tos);
  **        graphicsInitialised = true;
            break;

          case PVM.stopg:         // close graphics window
            Turtle.CloseGraphics();
  **        graphicsInitialised = false;
            break;


      } while (ps == running);
      if (ps != finished) PostMortem(results, pcNow);
  **  if (graphicsInitialised) {
  **    Console.Write("\nHit <Enter> to Exit!");
  **    Console.ReadLine();
  **    Turtle.CloseGraphics();
  **    graphicsInitialised = false;
      }
```

Alternatively, and perhaps more efficiently, each graphic opcode could have its own test, for example

```
            case PVM.turnt:         // turn turtle left
  **          if (!graphicsinitialised) ps = badGraph;
  **          else Turtle.TurnLeft(Pop());
              break;
```

QUESTION B13 **[24 marks]**

B13 The default location and orientation of the axes on the graphics canvas is bound to confuse people more familiar with axes where the y-axis points upwards. As a rather more interesting challenge, suppose the `InitGraphics` method were also to allow an alternative: `InitGraphics(width, height, x0, y0)`, whose effect would be to open a canvas (as before), but in this case arrange that subsequent calls to `Line(x1, y1, x2, y2)` would draw the specified line on a canvas whose **centre** corresponds to the point (`x0, y0`) and where the x- and y-axes pointed **right** and **up**. If the third and fourth arguments `x0` and `y0` are omitted, they should be taken to have the values `(0, 0)`.

This is not as difficult as it might at first appear, once it is realised that a simple linear transformation within the PVM of the values passed as arguments to `Line()` will produce the desired effect. To save you a lot of time, the required transformations have been explained in detail on page 8.

Modify the grammar and the PVM to accommodate the revised semantics of `InitGraphics`. [24 marks]

Firstly, one needs to add to the `InitGraphics` code to handle the two extra parameters (if present) and to stack up the two default zero values (if not):

```
    InitGraphicsStatement                      (. int expType; .)
    = "InitGraphics"
      "("        Expression<out expType>       (. if (!IsArith(expType))
                                                    SemError("arithmetic argument needed"); .)
        WEAK "," Expression<out expType>       (. if (!IsArith(expType))
                                                    SemError("arithmetic argument needed"); .)
  **    (  /* explicit at x0, y0 */
  **       WEAK "," Expression<out expType>    (. if (!IsArith(expType))
  **                                              SemError("arithmetic argument needed"); .)
  **       WEAK "," Expression<out expType>    (. if (!IsArith(expType))
  **                                              SemError("arithmetic argument needed"); .)
  **     | /* default at 0,0 */                (. CodeGen.LoadConstant(0);
  **                                              CodeGen.LoadConstant(0); .)
  **    )                                      (. CodeGen.InitGraphics(); .)
      ")"
      WEAK ";"
    .
```

The two arms of the PVM interpreter need extending too - care being taken to handle the order of popping the parameters off the run time stack. The intermediate variables `x0` and `y0` allow for ease of use in evaluating the parameters in the interpretation of `PVM.line`:

```
            case PVM.initg:         // initialize graphics window
  **          y0  = Pop();
  **          x0  = Pop();          // centre of window is conceptually (x0, y0)
              tos = Pop();          // height
              sos = Pop();          // width
  **          x0  = sos / 2 - x0;   // offsets for PVM.line
  **          y0  = tos / 2 + y0;
              Turtle.InitGraphics(sos, tos);
              graphicsInitialised = true;
              break;

            case PVM.line:          // draw line
  **          int y2 = y0 - Pop();
  **          int x2 = x0 + Pop();
  **          int y1 = y0 - Pop();
  **          int x1 = x0 + Pop();
              Turtle.Line(x1, y1, x2, y2);
              break;
```

QUESTION B14 (optional bonus) [8 marks]

B14 If, as suggested yesterday, you execute the sample programs in the kit using the system provided to you today you should notice that, while some "work" very well, some others - deceptively simple - seem to perform very badly (for example EG09, EG13 and EG15). Why do you suppose this is? Be assured that it is not the turtle logic that is incorrect; it must be something else. Can you suggest a fairly simple way to improve the system? [8 marks]

It will be interesting to see what students make of this one! The problem is that the PVM deals with integers only, while one really needs to be working with double or float values. However, there is no reason why the `TurtleLib` class cannot store the state of the turtle in `double` fields, and then do the truncation and rounding needed to deal with the screen coordinates when `DrawLine()` is called. The changes below make for a dramatic improvement.

```
     public class Turtle {

       static private double      // Turtle initial location, direction and pen setting
  **     direction = 0.0,
  **     turtleX   = 0.0,
  **     turtleY   = 0.0,
  **     homeX     = 0.0,
  **     homeY     = 0.0,
  **     homeD     = 0.0;

       // ..... omitted to save space

  **   public static void TurnLeft(double degrees) {
         direction -= degrees;
       } // Turtle.TurnLeft

  **   public static void Forward(double distance) {
  **     double rad  = 0.017453292 * direction;
  **     double newX = turtleX + distance * Math.Cos(rad);
  **     double newY = turtleY + distance * Math.Sin(rad);
  **     if (penDown) w.DrawLine((int) Math.Round(turtleX), (int) Math.Round(turtleY),
  **                             (int) Math.Round(newX),    (int) Math.Round(newY));
         turtleX = newX;
         turtleY = newY;
       } // Turtle.Forward

     } // class Turtle
```

Note: Added after a successful party. Some students doubted the value of the magic number 0.017453292 and, being mathematically inclined, substituted `Math.pi/180.0`. This is actually closer to 0.0174532925199 (difference in the 9th place) and they found to their delight that this fixed the problem for two of the "bad" examples. But it doesn't fix it for all of them - the suggestion above is still better, and would be even better still with the more accurate constant.

Morals of this story (a) I learn more from students than they think (b) roundoff errors in "real" computing can be a serious nuisance (c) testing an idea on only one example may not be the safest way to uncover funnies.

QUESTION B15 (optional bonus) [8 marks]

B15 Consider the program below. In C# and Parva, if a parameter is to be passed "by reference", the keyword `ref` is used to mark both the formal parameter and the corresponding actual argument, as shown in the first function call. In C, C++, Pascal and Modula-2 and various other languages, the equivalent of `ref` is used only on the formal parameter (as illustrated by the second function call).

Discuss whether it would be possible and/or preferable to use the "C" convention in Parva. If it is possible, explain how it could be done; if it is not possible, explain why not. [8 marks]

```
void Interchange(ref int x, ref int y) {
  int z = x; x = y; y = z;
} // Interchange

void Main() {
  int a = 56, b = 63;
  Interchange(ref a, ref b);   // C# and Parva style
  Interchange(a, b);           // C, Pascal, Modula-2 style
} // Main
```

This should sort out a few serious students: Of course one *can* write a grammar that omits the `ref` keyword on actual arguments. The problem then is that one has a very nasty LL(1) conflict, because in one case the actual argument must be a *Designator*, while in the other case it can be a general *Expression* - and a *Designator* is syntactically nothing more than a very simple form of *Expression*.

A little-exploited feature of Coco/R is that one can make use of so-called "resolvers" in situations like this, which allow one to drive a parse "semantically" in situations where syntax on its own fails. I have not used this in this course before 2015, because all the examples I have ever used prior to that can be resolved by careful factorisation. However, this year the feature was mentioned, and this problem was in fact discussed in the little videos made to supplement the discussion of prac 7 and the material in Chapter 14. It was also illustrated in the prologue to Section B of this examination.

The solution below shows how it can be done. In fact it goes further, and leaves the option of adding `ref` as well!

One can comment that the C# style is highly commendable. It is little effort to add the `ref` in both places, and it gives a useful check agains errors that might otherwise go undetected when simple expressions masquerade as designators and vice versa.

So the rigorous code provided to the candidates

```
     OneArg<Entry fp>                    (. int argType;
                                            DesType des; .)
     =  (  Expression<out argType>       (. if (fp != null) {
                                              if (!Compatible(argType, fp.type))
                                                SemError("argument type mismatch");
  **                                          if (fp.byRef)
  **                                            SemError("this argument must be passed by reference");
                                            } .)
        |
           "ref" Designator<out des>     (. if (fp != null) {
                                              if (!Compatible(des.type, fp.type))
                                                SemError("argument type mismatch");
  **                                          if (!fp.byRef)
  **                                            SemError("this argument must be passed by value");
                                            } .)
        )
     .
```

can be replaced by:

```
     OneArg<Entry fp>                    (. int argType;
                                            DesType des; .)
  ** =  (  IF (fp != null && !fp.byRef)
           Expression<out argType>       (. if (fp != null) {
                                              if (!Compatible(argType, fp.type))
                                                SemError("argument type mismatch");
                                            } .)
        |
  **       [ "ref" ] Designator<out des>  (. if (fp != null) {
                                              if (!Compatible(des.type, fp.type))
                                                SemError("argument type mismatch");
                                            } .)
        )
     .
```

Notice that one still needs the tests for a non-null argument (which might manifest itself if the number of formal and actual parameters were, incorrectly, different).