RHODES UNIVERSITY

June Examinations - 2014

Computer Science 301 - Paper 2 (Solutions)

Examiners:	Time 4 hours
Prof P.D. Terry	Marks 180
Prof P. Blignaut	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 class definitions 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 [95 marks]

A1. (a) What distinguishes a "compiler" from an "assembler"? [2]

A compiler translates "high level" code to object/machine code; an assembler translates low-level code to object/machine code.

(b) How would you explain and distinguish the terms "self-resident compiler", "cross-compiler" and "self-compiling compiler" to a student taking a course in compiler constuction for the first time, and how could you lead them to believe that a self-compiling compiler can ever be a reality? [10]

Self-resident - the compiler runs on the same machines as it generates code for.

Cross-compiler - the compiler generates code for a different machine from the one on which it executes.

Self-compiling compiler - source for the compiler is available in the language that is to be compiled. This implies that the compiler should be able to replicate its own object/machine code. Development of such compilers requires that they first be implemented in some other host language for which a compiler already exists, and are then hand-translated into a version written in the source code (itself a classical bootstrap situation).

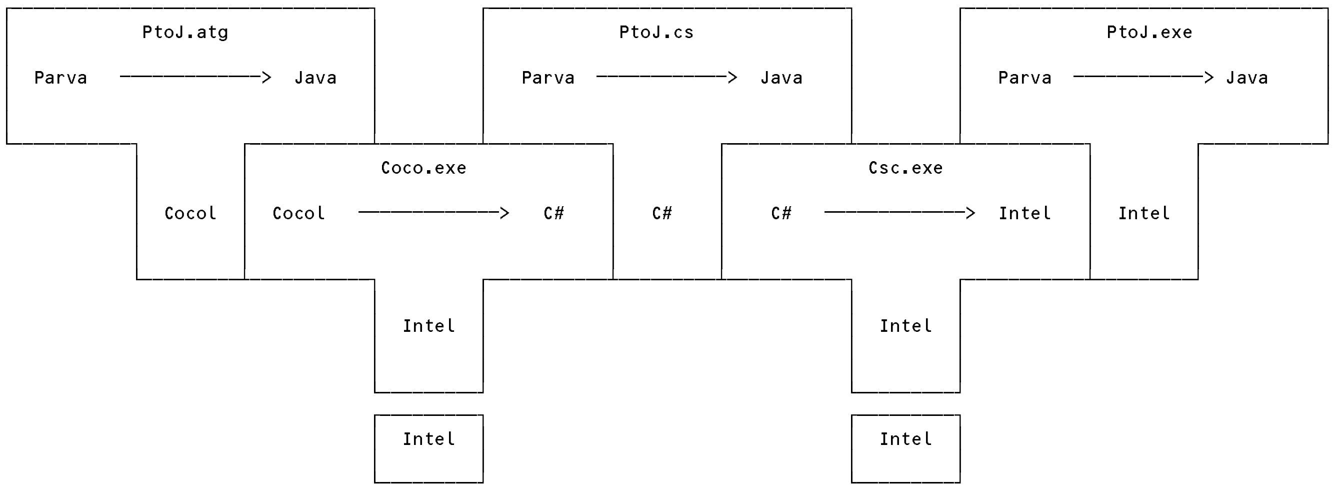
A2. Explain clearly and simply what you understand by the terms syntax, static semantics, and dynamic semantics, and what distinguishes one from another. [8]

Syntax rules describe the arrangement or form that program source, statements and expressions can take or in which they can be combined (for example, in Java source code a while statement requires a parenthesized expression and another statement to follow the parentheses). Static semantic rules relate to the way in which types, expressions identifiers and so on interact (for example, in a while statement the parenthesized expression must be one composed of operands and operators that not only satisfy syntactic rules, but which also evaluates to a Boolean result). Dynamic semantics describe what happens when the program (or statement) executes (for example, in the case of a while statement the subsidiary statement will be executed, possibly repeatedly, for as long as a re-evaluation of the expression yields a Boolean value of **true***).*

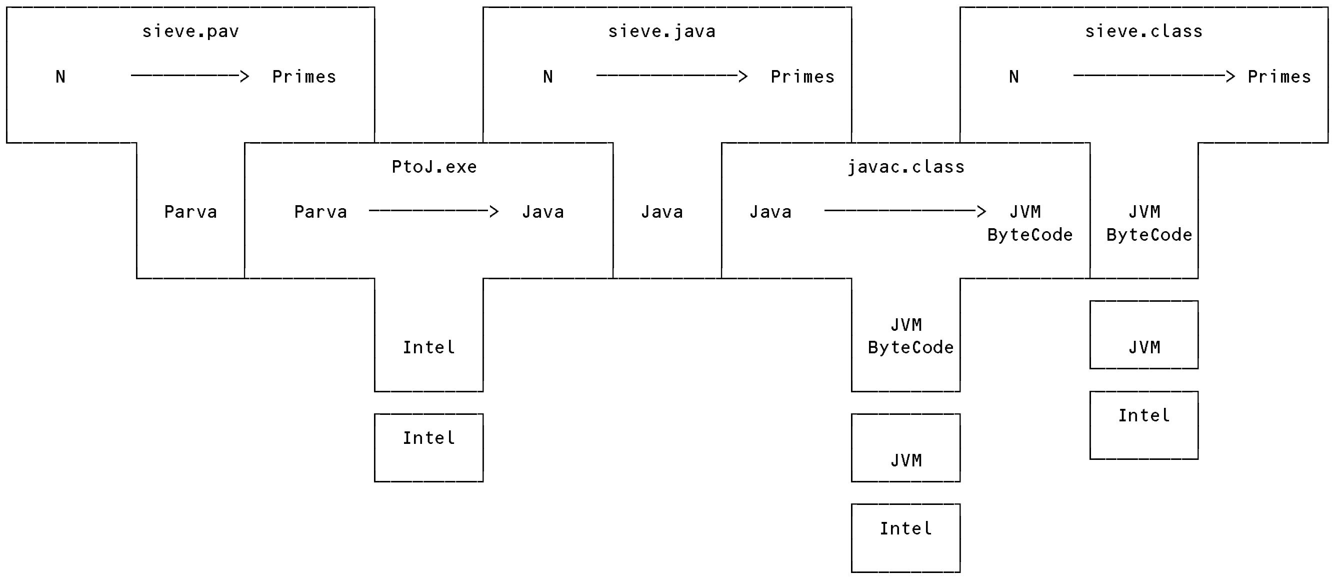
A3. In the practical sessions you should have used our Parva to Java translator. This was developed in C# from an attributed grammar for Parva written in Cocol. The Coco/R system used this grammar to generate C# source code that was compiled by a software development kit for C# to yield an EXE file that was circulated to your class. Draw T-diagrams showing the process used to produce this system, and go on to draw T-diagrams showing how you managed to take a program `SIEVE.PAV`, convert it to Java and then run it on the PC using the Java compiler `javac.class` and a JVM hosted on the PC system as your system of choice. [10]

A set of blank T-diagrams is provided in the free information, which you can complete and submit with your answer book.

The ParvaToJava program was specified in Cocol, converted to C# by Coco/R and compiled:



Using ParvaToJava can be depicted as follows:



A4. (a) What would you understand by a statement from a group of very unhappy students who came to you and told you that they had discovered that their carefully thought out grammar had turned out to be "ambiguous"? [2]

An ambiguous grammar is one where at least one sentence of the language it describes can be derived in more than one way, that is, one can find more than one parse tree representing that sentence.

(b) Why can an LL(1) conformant grammar never be ambiguous? (Explain carefully!) [3]

Because an LL(1) grammar has the property that parsing can only proceed on the basis of a single look ahead, and this path will thus always be uniquely defined.

(c) Does it then follow that if a grammar is not ambiguous it must automatically be LL(1) conformant? Support your argument by giving some simple example grammars. [6]

No, it does not. A typical example of a non-LL(1), unambiguous grammar is the classic left recursive one for simple expressions

```
Goal       = Expression .
Expression = Term | Expression "-" Term | Expression "+" Term .
Term       = Factor | Term "*" Factor | Term "/" Factor .
Factor     = "a" | "b" | "c" | "d" .
```

(d) Discuss the anomaly in programming language design that gives rise to the so-called "dangling else" problem. Show how one could easily design a language that does not suffer from this problem. Also explain why the problem is in any case less severe than it might at first appear. [6]

The "dangling else" is the ambiguity that arises when one wishes to define the if-else statement as

IfElseStatement = "if" "(" Condition ")" Statement ["else" Statement] .

which allows a construct like

"if" (boolExp1) "if" (boolExp2) statement1 "else" statement2

to be derived in two different ways, corresponding either to

"if" (boolExp1) { "if" (boolExp2) statement1 } "else" statement2

or

"if" (boolExp1) { "if" (boolExp2) statement1 "else" statement2 }

The problem is easily eliminated if one insists on a closing keyword:

IfElseStatement = "if" "(" Condition ")" Statement ["else" Statement] "endif".

But it is not really a problem, as a recursive descent parser will naturally bind the "else" clause to the most recent "if" and produce the semantic effect that is needed.

A5. The Cocol grammar below attempts to describe declarations in a simple C-like language:

```
COMPILER Declarations

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

TOKENS
  ident  = letter { letter | digit | "_" ( letter | digit ) } .

COMMENTS FROM "/*" TO "*/"

IGNORE CHR(0) .. CHR(31)

PRODUCTIONS
  Declarations = { DecList } EOF .
  DecList      = Type OneDecl { "," OneDecl } ";"  .
  Type         = "int" | "void" | "bool" | "char" .
  OneDecl      = "*" OneDecl | Direct .
  Direct       = ( ident | "(" OneDecl ")" ) [ Params ] .
  Params       = "(" [ OneParam { "," OneParam } ] ")" .
  OneParam     = Type [ OneDecl ] .
END Declarations.
```

(a) Is this an LL(1) compliant grammar? Explain your reasoning in some detail. [8]

Yes it is. The traditional full analysis might proceed by transforming the grammar to one with no meta-brackets. This is straightforward but a bit tedious:

```
Declarations     =  DeclarationSeq EOF .
DeclarationSeq   =  DecList DeclarationsSeq | .
DecList          =  Type OneDecl MoreDecls ";" .
Type             =  "int" | "void" | "bool" | "char" .
OneDecl          =  "*" OneDecl | Direct .
MoreDecls        =  "," OneDecl MoreDecls | .
Direct           =  ( ident | "(" OneDecl ")" ) OptParams .
```

```
OptParams        =  "(" ParamList ")" | .
ParamList        =  OneParam MoreParams | .
MoreParams       =  "," OneParam MoreParams | .
OneParam         =  Type OptDecl .
OptDecl          =  OneDecl | .
```

For Rule 1 we have only to consider the productions for Type and for Direct, which clearly cause no trouble. For Rule 2 we discern that the nullable non-terminals now are

```
DeclarationsSeq, MoreDec s, OptParams, ParamList, MoreParams, OptDecl
```

but these cause no trouble, as can be seen from their FIRST and FOLLOW sets

```
DeclarationsSeq
first:   "int" "void" "bool" "char"    follow:  EOF

MoreDecls
first:   ","                           follow:  ";"

OptParams
first:   "("                           follow:  ";" "," ")"

ParamList
first:   "int" "void" "bool" "char"    follow:  ")"

MoreParams
first:   ","                           follow:  ")"

OptDecl
first:   ident "*" "("                 follow:  "," ")"


Direct
first:   ident "("                     follow:  ";" "," ")"

OneParam
first:   "int" "void" "bool" "char"    follow:  "," ")"
```

However, this grammar is just as easily analysed in the original EBNF form

```
Declarations = { DecList } EOF .
DecList      = Type OneDecl { "," OneDecl } ";"  .
Type         = "int" | "void" | "bool" | "char" .
OneDecl      = "*" OneDecl | Direct .
Direct       = ( ident | "(" OneDecl ")" ) [ Params ] .
Params       = "(" [ OneParam { "," OneParam } ] ")" .
OneParam     = Type [ OneDecl ] .
```

For Rule 1 we have only to consider the productions for Type and for Direct, which clearly cause no trouble. For Rule 2 we consider the nullable portions, which are

```
{ Declist }   and   [ Params ]   and   [ OneParam { "," OneParam } ]
and { "," OneDecl }  and  [ OneDecl ]

FIRST(  {DecList} )  = FIRST(Type) = { "int", "void", "bool", "char" }
FOLLOW( {DecList} ) = { EOF }

FIRST(  [ Params] ) = { "(" }
FOLLOW( [ Params ] ) = FOLLOW(Direct) = FOLLOW(OneDecl) = { "," , ")" }

FIRST(  [ OneParam { "," OneParam } ] ) = FIRST(OneParam) = FIRST(Type)
FOLLOW( [ OneParam { "," OneParam } ] ) = { ")" }

FIRST(  { "," OneParam } ) = { "," }
FOLLOW( { "," OneParam } ) = { ")" }

FIRST(  { "," OneDecl } ) = { "," }
FOLLOW( { "," OneDecl } ) = { ";" }
```

```
FIRST(  [ OneDecl ] ) = { "*", ident "(" }
FOLLOW( [ OneDecl ] ) = { FOLLOW( OneParam ) = { "," , ")" .
```

and these clearly satisfy Rule 2.

(b) 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, identSym, intSym, voidSym, boolSym, charSym, commaSym, lparenSym,
rparenSym, semicolonSym, starSym;
```

How would you complete the parser routines below? [20]

A spaced copy of this system appears in the free information, which you are invited to complete and hand in with your answer book.

This is straightforward but a bit time-consuming. Students have seen several examples like this, and generally master them pretty well

```
static IntSet
  FirstDeclarations = new IntSet(intSym, voidSym, boolSym, charSym),
  FirstDeclList     = new IntSet(intSym, voidSym, boolSym, charSym),
  FirstType         = new IntSet(intSym, voidSym, boolSym, charSym),
  FirstOneDecl      = new IntSet(starSym, identSym, lparenSym),
  FirstDirect       = new IntSet(identSym, lparenSym),
  FirstParams       = new IntSet(lparenSym),
  FirstOneParam     = new IntSet(intSym, voidSym, boolSym, charSym),

static void Declarations () {
// Declarations = { DecList } EOF .
  while FirstDeclList.contains(sym.kind) DecList();
  accept(EOFSym, "EOF expected");
}

static void DecList () {
// DecList = Type OneDecl { "," OneDecl } ";"  .
  Type();
  OneDecl();
  while (sym.kind == commaSym) {
    getSym(); OneDecl();
  }
  accept(semicolonSym, "; expected");
}

static void Type () {
// Type = "int" | "void" | "bool" | "char" .
  if Firsttype.contains(sym.kind) getSym();
  else abort("invalid type");
}

static void OneDecl () {
// OneDecl = "*" OneDecl | Direct .
  if (sym.kind == starSym) {
    getSym(); oneDecl();
  }
  else if (FirstDirect.contains(sym.kind)) Direct();
  else abort("invalid start to OneDecl");
}

static void Direct () {
// Direct = ( ident | "(" OneDecl ")" ) [ Params ] .
  if (sym.kind == identSym()) getSym;
  else if (sym.kind == lparenSym) {
    getSym;
    OneDecl();
    accept(rparenSym, ") expected");
  }
  else abort("invalid start to Direct");
  if (sym.kind == lparenSym) Params();
}
```

```
static void Params () {
// Params = "(" [ OneParam { "," OneParam } ] ")" .
  accept(lparenSym, "( expected");
  if (FirstOneParam.contains(sym.kind)) {
    OneParam();
    while (sym.kind == commaSym) {
      getSym(); OneParam();
    }
  }
  accept(rparenSym, ") expected");
}

static void OneParam () {
// OneParam  = Type [ OneDecl ] .
  Type();
  if (FirstOneDecl.contains(sym.kind)) OneDecl();
}
```

A6. The examination results for a class of students are to be supplied in a file which gives, for each student, their student number, surname, gender, faculty, and mark obtained - a typical extract might read

```
63T0844  Terry           Male   S  85
12M1234  MacDonald       Female H  50.00
12O1234  O'Malley        Male   C  78.6
11S1234  Smith-Jenkins   Male   S  55.67
81W4251  Bradshaw        Female S  99
09F4567  Van Wyk Smith   Female C  48
01M1234  MacHine         Male   L  10.30
```

Student numbers are of a standard format - two digits giving the year of first registration (02 denotes 2002, 95 denotes 1995 and so on), followed by the initial letter of the surname at the time of registration and then by a sequence of four digits. Names begin with a capital letter, and composite surnames like "Van Wyk Smith" and ones with embedded apostrophes and hyphens are also permissible. The faculty is denoted by a single letter with an obvious relationship to our faculties of Science, Humanities, Commerce, Law, Education and Pharmacy.

(a) Show how an attributed Cocol grammar could be used to parse such a file and determine the number of students in the list who first registered more than three years ago. A skeleton grammar file is provided in the free information, which you could complete and submit with your answer book. [15]

Again, this is pretty straightforward and students have had experience of several similar grammars. The sort of solution I expected to receive is as follows (a Java version):

```
import Library.*;

COMPILER Marks $NC /* token names, generate compiler driver */
/* P.D. Terry, Rhodes University, 2014 */

static int
  total = 0,
  longer = 0;

CHARACTERS
  lf            = CHR(10) .
  control       = CHR(0) .. CHR(31) .
  letter        = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .
  uletter       = "ABCDEFGHIJKLMNOPQRSTUVWXYZ" .
  digit         = "0123456789" .
  printable     = ANY - control .

TOKENS
  name          = uletter { "'" uletter | "-" uletter | letter } .
  studentNumber = digit digit uletter digit digit digit digit .
  mark          = digit { digit} [ "." digit { digit } ] .
  eol           = lf .

IGNORE  CHR(9) + CHR(11) .. CHR(13)
```

```
PRODUCTIONS

  Marks
  = {
      StudentNumber
      FullName
      Gender
      Faculty
      mark SYNC eol   (. total++; .)
    }               (. IO.writeLine("Total students: " + total);
                       IO.writeLine(longer +
                                    " of these first registered more than 3 years ago"); .)
  .

  FullName
  = name { name } .

  Gender
  = "Male" | "Female" .

  Faculty
  =  "S" | "H" | "C" | "E" | "L" | "P" .


  StudentNumber      (. int year; .)
  = studentNumber    (. String s = token.val.substring(0, 2);
                        try {
                          year = Integer.parseInt(s);
                        } catch (NumberFormatException e) {
                          year = 0; SemError("number too large");
                        }
                        if (year <= 14) year = 2000 + year; else year = 1900 + year;
                        if (2014 - year >= 3) longer++; .)
  .

END Marks.
```

However, a submission from a student some years ago when I used this example suggested that one only really needs to look at the first two characters on each line. Correctly developed this is reflected in code like the following (a C# version). (As it happened the student's submission was incorrect - he forgot that one would still have to read the superfluous material on each line, which is why the `eol` *token has a rather odd definition).*

```
using Library;

COMPILER Marks1 $NC /* token names, generate compiler driver */
/* P.D. Terry, Rhodes University, 2014 */

static int
  total = 0,
  longer = 0;

CHARACTERS
  lf            = CHR(10) .
  cr            = CHR(13) .
  control       = CHR(0) .. CHR(31) .
  uletter       = "ABCDEFGHIJKLMNOPQRSTUVWXYZ" .
  digit         = "0123456789" .
  printable     = ANY - control .

TOKENS
  studentNumber = digit digit .
  eol           = uletter { printable } [ cr] lf .

IGNORE  CHR(9) + CHR(11) .. CHR(13)

PRODUCTIONS

  Marks1
  = { studentNumber   (. int year;
                         try {
                          year = Convert.ToInt32(token.val);
                        } catch (Exception) {
                          year = 0; SemError("invalid number");
                        }
                        if (year <= 14) year = 2000 + year; else year = 1900 + year;
                        if (2014 - year >= 3) longer++; .)

      SYNC eol        (. total++; .)
    }                 (. IO.WriteLine("Total students: " + total);
                         IO.WriteLine(longer + " of these first registered more than 3 years ago"); .) .

END Marks1.
```

(b) Clearly this problem can be solved very easily without the use of a tool like Coco/R. Are there any advantages to be gained from using Coco in simple applications of this sort, and if so, what might these advantages be? [5]

The advantages of using Coco are that the parser and scanner are generated very easily from a rigorous formal specification, and the parser can also incorporate error checking very easily rather than requiring a programmer to hack away (or more than likely, duck error checking completely!)

Section B [85 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 EXAMx.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 accept simple class type definitions, and allow programs to declare and use variables of those 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. Continue now to answer the following unseen questions, all but the last of which should require no changes to the code generator or interpreter.

QUESTION B7 **[9 marks]**

B7. The semantic checking in the supplied system is incomplete. For example, the compiler will accept code like the following without complaint:

```
class Wrong {
  int x;
  bool x;
};

Wrong silly;
silly.z = silly.z;
```

Indicate how and where such oversights might be corrected.

```
      OneField<. Entry classEntry, int type, List<Entry> fieldList .>
                                              (. Entry field = new Entry();
                                                 field.kind = Kinds.Var;
                                                 field.type = type;
                                                 field.fieldList = fieldList;
                                                 field.offset = classEntry.fields;
                                                 field.declared = true; .)
  **  = Ident<out field.name>                 (. foreach (Entry e in classEntry.fieldList)
  **                                               if (e.name.Equals(field.name))
  **                                                 SemError("field name already in scope");
                                                 classEntry.fieldList.Add(field);
                                                 classEntry.fields++; .) .


      Designator<out DesType des>             (. string name;
                                                 int indexType; .)
      =  Ident<out name>                      (. Entry entry = Table.Find(name);
                                                 if (!entry.declared)
```

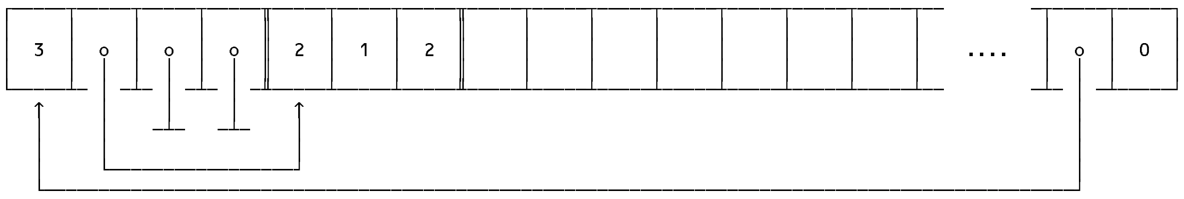
```
                                                  SemError("undeclared identifier");
                                                  des = new DesType(entry);
                                                  if (entry.kind == Kinds.Var) CodeGen.LoadAddress(entry); .)
       {   ( "["                               (. if (IsArray(des.type)) des.type--;
                                                  else SemError("unexpected subscript");
                                                  if (des.entry.kind != Kinds.Var)
                                                    SemError("unexpected subscript");
                                                  CodeGen.Dereference(); .)
                Expression<out indexType>      (. if (!IsArith(indexType))
                                                    SemError("invalid subscript type");
                                                  CodeGen.Index(); .)
             "]"
           )
         |
           (  "." Ident<out name>              (. CodeGen.Dereference();
                                                  Entry fieldEntry = Table.FindField(des.entry, name);
                                                  if (fieldEntry != null) {
                                                    des.entry = fieldEntry;
                                                    des.type  = fieldEntry.type;
                                                    CodeGen.LoadConstant(des.entry.offset);
                                                    CodeGen.Index();
                                                  }
**                                                else SemError("unexpected field"); .)
           )
       } .
```

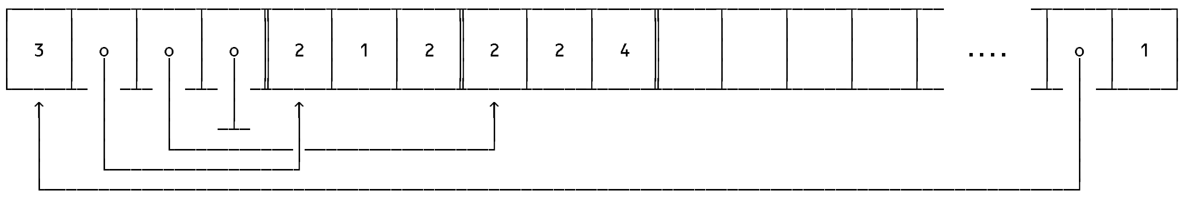
QUESTION B8 **[12 marks]**

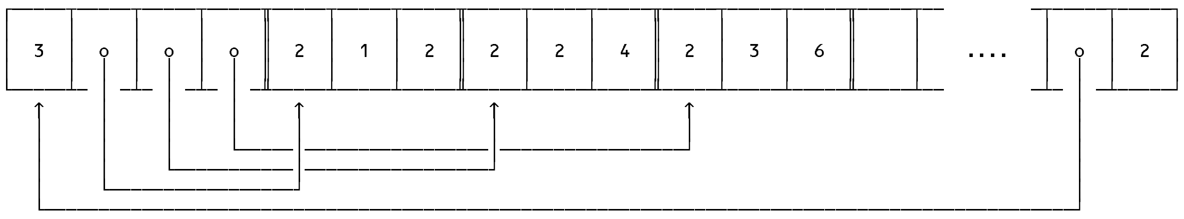
B8. The following is a complete, if slightly pointless, program. For each iteration of the *while* loop, illustrate the state of the *stack* and *heap* immediately before the `i++;` assignment at point (a) is *executed:*

```
void main() { // B8.pav
  class One {
    int x,y;
  };

  int i = 0;
  One[] list = new One[3];
  while (i <= 3) {
    int j = i + 1;
    list[i] = new One();
    list[i].x = j;
    list[i].y = 2 * j;
    i++;                    // point (a)
  } // while
  // point (b)
} // main
```



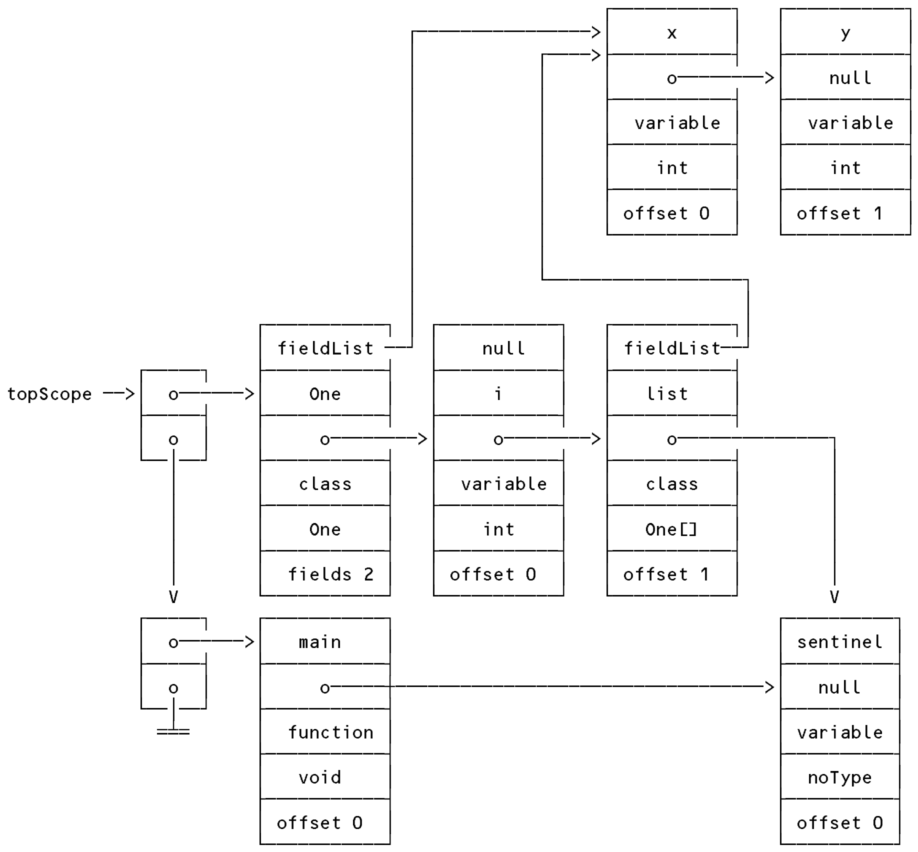




QUESTION B9 **[14 marks]**

B9. For the program in the previous question, draw a diagram showing the major components of the symbol table as it would exist at point (b), after completing *compilation* of the *while* loop.

Something like this is required. Note that the variable `j` *has gone out of scope again at this point.*



QUESTION B10 **[8 marks]**

B10. Modify the compiler so that objects of a class may be declared immediately following the class definition as exemplified by

```
class One {
  int x, y;
} a, b = new One(), c = b, d = null;
```

```
      ClassDeclaration<StackFrame frame>        (. Entry classEntry = new Entry(); .)
      = "class" Ident<out classEntry.name>      (. classEntry.type = Types.AddType(classEntry.name);
                                                   classEntry.kind = Kinds.Class;
                                                   classEntry.fields = 0;
                                                   classEntry.fieldList = new List<Entry>();
                                                   Table.Insert(classEntry); .)
         "{"
            { FieldList<classEntry> WEAK ";"
            }
         "}"                                    (. /* Table.Insert(classEntry); */ .)
**       [ ObjList<frame, new DesType(classEntry) >
**       ]
         WEAK ";"
      .
```

QUESTION B11 **[8 marks]**

B11. The compiler will accept the definition of a class with no fields at all, such as

```
class Empty {
  // no field list
};
```

This might appear strange, and even stranger when it is pointed out that one can compile and execute a statement like

```
Empty e, f = e;
```

However, although compilation of a similar statement

```
Empty e = new Empty(), f = e;
```

will raise no alarm bells, the run time execution of this statement would result in an error.

Presumably this reflects a blemish in language design or implementation. Suggest steps that could be taken to improve the language specification and/or the compilation process.

There are several possibilities. The error is trapped at run time by the interpreter's handling of the `PVM.anew` *opcode. There is no real reason to forbid the definition of an empty record, but one should at least issue a warning, which could be done as follows*

```
      ClassDeclaration<StackFrame frame>        (. Entry classEntry = new Entry(); .)
      = "class" Ident<out classentry.name>      (. classEntry.type = Types.addtype(classentry.name);
                                                   classEntry.kind = Kinds.class;
                                                   classEntry.fields = 0;
                                                   classEntry.fieldlist = new list<entry>(); .)
        "{"
           { FieldList<classentry> weak ";"
           }
        "}"                                     (. table.Insert(classentry); .)
**                                                 if (classEntry.fields == 0)
**                                                   if (warnings) Warning("empty class"); .)
        [ ObjList<frame, new DesType(classEntry) >
        ]
        WEAK ";" .
```

and, of course, the `SemError` *method could be called if one wanted to veto the practice.*

One might wonder at whether empty class definitions could serve any purpose. Indeed they might - in "frame" files for example, where the fields might be added to the system by a program generator in some situations but not others. But it would be too much to expect candidates to dream that one up!

QUESTION B12 [8 marks]

B12. The following program illustrates a classic way to construct and display a simple stack:

```
void main() {  // B12.pav

  class Node {
     Node link;        // point (a)
     int value;
  };

  Node list = null;
  int data;
  read(data);
  while (data != 0) {
    Node next = new Node();
    next.value = data;
    next.link = list;
    list = next;
    read(data);
  }

  while (list != null) {
    writeLine(list.value);
    list = list.link;
  }

} // main
```

Unfortunately the compiler will not accept it, as reference is made to `Node` at point (a) before the definition of this class is completed. However, a trivial change to the compiler will overcome this problem. Show the change, and then comment on how it is that this violation of "must define before use" can work when others cannot as easily be made to work on a single pass, if at all.

All that is needed is to make the entry into the symbol table before parsing the field list:

```
      ClassDeclaration<StackFrame frame>        (. Entry classEntry = new Entry(); .)
      = "class" Ident<out classEntry.name>      (. classEntry.type = Types.AddType(classEntry.name);
                                                   classEntry.kind = Kinds.Class;
                                                   classEntry.fields = 0;
                                                   classEntry.fieldList = new List<Entry>(); .)
**                                                 Table.Insert(classEntry); .)
         "{"
            { FieldList<classEntry> WEAK ";"
            }
         "}"                                    (. if (classEntry.fields == 0)
                                                     if (warnings) Warning("empty class"); .)
        [ ObjList<frame, new DesType(classEntry) >
        ]
        WEAK ";" .
```

This works because an object is implemented via a pointer, and any subtler details of the pointer - apart from its existence - will be needed only after the entry is made. At the point where it is entered above we already know the type and kind of the `ClassEntry` *node!*

QUESTION B13 [11 marks]

B13. Consider another simple, if rather awkward, program

```
void main() {  // B14.pav

 class TestResult {
   int mark;
   bool passed;
 };

 int n = 0;

 while (true) {
   TestResult t = new TestResult();
   read("Supply mark (negative exits ", t.mark);
   if (t.mark < 0) break;
   t.passed = t.mark >= 50;
   n++;
 }
 writeLine(n, " candidates wrote the test");
} // main
```

Comment on the run-time behaviour of this program, in particular with respect to its use of the heap [4]. How, if at all, would this behaviour alter if the program were rewritten as follows [4]

```
void main() {  // B13a.pav

 class TestResult {
   int mark;
   bool passed;
 };

 TestResult t = new TestResult();
 int n = 0;

 while (true) {
   read("Supply mark (negative exits ", t.mark);
   if (t.mark < 0) break;
   t.passed = t.mark >= 50;
   n++;
 }
 writeLine(n, " candidates wrote the test");
} // main
```

and how, if at all, would the behaviour differ from what you could expect were an equivalent program to be written in C# or Java? What, in particular, do C# and Java provide that your Parva implementation does not provide? [3]

The insight that is being sought here is, of course, that in the first example a new object is allocated on the heap on each pass through the loop, which has the effect of turning the previous object into "garbage" which cannot be "collected" as it would be in C# or Java. In the second example the same object is, of course, simply re-used.

QUESTION B14 **[15 marks]**

B14. Consider the program below. Explain why the output should be "false false" and not "false true" [3].

```
  void main() {  // B14.pav

    class One {
      int x, y;
    } a = new One(); b = new One();

    a.x = 1;   a.y = 2;
    b.x = a.x; b.y = 3;

    writeLine(a == b);
    b.y = a.y;
    writeLine(a == b);
  } // main
```

The implementation of the `==` operation will be accepted without complaint, and will compare the values of the links to the objects `a` and `b`, which are not altered by the assignments to their fields. Students should hopefully have been aware of this since their first year, but I guess not all will be.

Now go in to modify the language to introduce a function that allows you to compare two objects in the way a user might expect, along the following lines: [12]

```
  void main() {  // B14a.pav

    class One {
      int x, y;
    } a = new One(); b = new One();

    a.x = 1;   a.y = 2;
    b.x = a.x; b.y = 3;

    writeLine(isEqual(a, b));  // false
    b.y = a.y;
    writeLine(isEqual(a, b));  // true
  } // main
```

Hint: feel free to modify the code generator and interpreter to accomplish this task.

This requires more code to be written than in the other exercises, and it may take time to get this right, even though parts of it are similar to the implementation of the `size()` *function that was drawn to students' attention when the second examination kit was released. We require a modification to the* `Primary` *non-terminal:*

```
Primary<out int type>                (. type = Types.noType;
                                        int size;
                                        string name;
                                        DesType des, des1;
                                        Entry entry;
                                        ConstRec con; .)
=  ....
   | "size" "(" Designator<out des>  (. if (des.entry.kind != Kinds.Var)
                                          SemError("variable expected");
                                        if (!IsArray(des.type) && !IsClass(des.type))
                                          SemError("array or object expected");
                                        CodeGen.Dereference();
                                        CodeGen.Dereference();
                                        type = Types.intType; .)
     ")"
```

```
**     | "isEqual" "("
**         Designator<out des>             (. if (des.entry.kind != Kinds.Var)
**                                              SemError("variable expected");
**                                            if (!IsArray(des.type) && !IsClass(des.type))
**                                              SemError("array or object expected");
**                                            CodeGen.Dereference(); .)
**         "," Designator<out des1>        (. if (des1.entry.kind != Kinds.Var)
**                                              SemError("variable expected");
**                                            if (!IsArray(des1.type) && !IsClass(des1.type))
**                                              SemError("array or object expected");
**                                            if (des.entry.type != des1.entry.type)
**                                              SemError("incomparable operand types");
**                                            CodeGen.Dereference();
**                                            CodeGen.CompareHeapObjects();
**                                            type = Types.boolType; .)
**       ") .
```

To the code generator must be added:

```
public static void CompareHeapObjects() {
// Generates code to pop two addresses from evaluation stack
// and push Boolean value A == B where A and B are objects or
// arrays allocated on the heap
  Emit(PVM.equal);
} // CodeGen.CompareHeapObjects
```

and to the interpreter's large switch statement must be added the clause:

```
case PVM.equal:         // compare heap allocated objects for equality
  adr  = Pop();
  adr1 = Pop();
  if (adr * adr1 == 0) ps = nullRef;
  if (mem[adr1] != mem[adr])
    Push(0);            // unequal sizes => unequal
  else {
    loop = mem[adr];
    while (loop > 0 && mem[adr1 + loop] == mem[adr + loop]) loop--;
    Push(loop == 0 ? 1 : 0);
  }
  break;
```

and of course there are other cosmetic changes to the initialisation of the interpreter.

Section C

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

Candidates were provided with the basic ideas adding simple class definitions to the Parva language and were invited to modify a supplied Parva implementation to acccommodate these.

They were provided with an exam kit for Java or C#, containing a working Parva compiler like that which they had used in the practical course. They were also given a suite of simple, suggestive test programs. Finally, 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 Parva 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; 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 encouraaged 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 naïve in some respects, in order to lay the ground for the unseen questions of the following day.