

RHODES UNIVERSITY

November Examinations - 2017

Computer Science 301 - Paper 2 - Compilers - Solutions

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

Time 4 hours
Marks 180
Pages 16 (please check!)

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

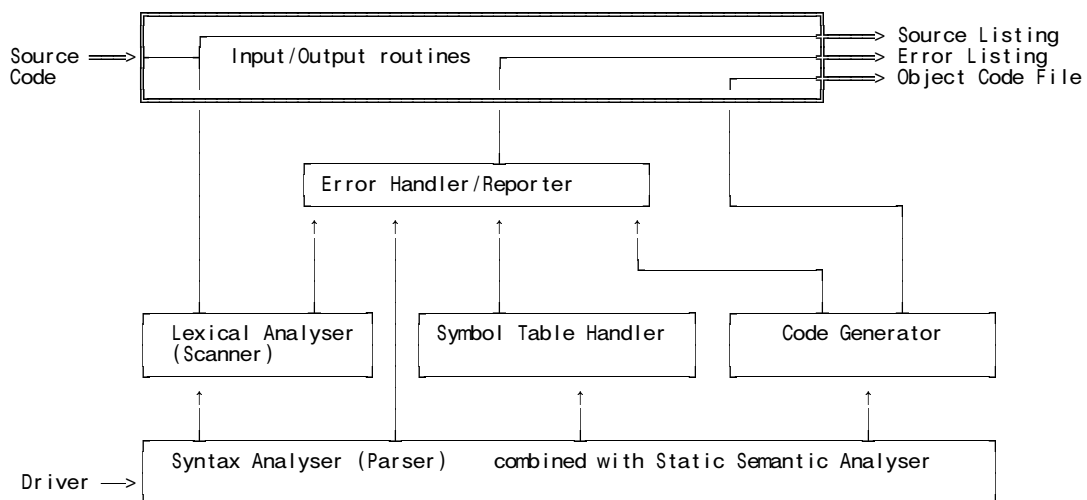
(For the benefit of future readers of this paper, various free information was made available to the students 24 hours before the formal examination. This included an augmented version of "Section C" - a request to devise a simple Boolean expression compiler targetting the Parva Virtual Machine interpreter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

Section A [90 marks]

QUESTION A1

[8 marks]

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



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

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

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

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

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

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

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

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

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

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

$$(1) \text{ FIRST}(\sigma) \quad \text{where } \sigma \in (N \cup T)^+$$

$$a \in \text{FIRST}(\sigma) \quad \text{if } \sigma \Rightarrow^* a\tau \quad (a \in T; \sigma, \tau \in (N \cup T)^+)$$

$$(2) \text{ FOLLOW}(A) \quad \text{where } A \in N$$

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

(3) A context-free grammar

All productions are of the form $\alpha \rightarrow \beta$ where $\alpha \in N, \beta \in (N \cup T)^*$

(4) A reduced grammar

A context-free grammar is said to be reduced if, for each non-terminal B we can write

$$S \Rightarrow^* \alpha B \beta$$

for some strings α and β (B is *reachable*), and where

$$B \Rightarrow^* \gamma$$

for some $\gamma \in T^*$. (B can be derived to a *sentence*)

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

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

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

Rule 1 :

$$\text{FIRST}(\xi_{i1}) \cap \text{FIRST}(\xi_{i2}) \cap \text{FIRST}(\xi_{i3}) \cap \dots \cap \text{FIRST}(\xi_{in}) = \emptyset$$

Rule 2 :

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

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

A better answer would be as follows

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

$$A_i \rightarrow \xi_{i1} \mid \xi_{i2} \mid \dots \mid \xi_{in}$$

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 (**Rule 1**)

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

and, in addition (**Rule 2**)

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

$$A_i \rightarrow \xi_{i1} \mid \xi_{i2} \mid \dots \mid \xi_{in}$$

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 $\text{FOLLOW}(A_i)$ of symbols that may follow any sequence generated from A_i , that is

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

or, rather more loosely,

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

One does not form one intersection of all the FIRST sets; and Rule 2 only applies when A is nullable. These are both commonly misunderstood by students.

- (c) Describe the language generated by the following grammar, using English or simple mathematics. [2 marks]

$$\begin{aligned} S &\rightarrow A B \\ A &\rightarrow a A \mid a \\ B &\rightarrow b B c \mid bc \end{aligned}$$

$$L = \{ a^m b^n c^n \mid m > 0, n > 0 \}$$

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

No it is not. The productions for A and B both break Rule 1. LL(1) grammars are easily written. One is:

$$\begin{aligned} S &\rightarrow A B \\ A &\rightarrow a \{ a \} \\ B &\rightarrow b \{ B \} c \end{aligned}$$

- (e) The following grammar describes strings comprised of an equal number of the characters a and b , terminated by a period, such as $aababbba$. Is it an LL(1) grammar? Explain. [2 marks]

$$\begin{aligned} S &\rightarrow B . \\ B &\rightarrow a B b B \mid b B a B \mid \varepsilon \end{aligned}$$

No it is not. B is nullable, and $\text{FIRST}(B) = \{ a, b \} = \text{FOLLOW}(B)$ so Rule 2 is broken

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

```
int count = 0;
char ch = IO.ReadChar();
while (ch != '.') {
    if (ch == 'a') count++;
    else if (ch == 'b') count--;
    else Abort("invalid string");
    ch = IO.ReadChar();
}
if (count == 0) IO.Write("valid string"); else IO.Write("invalid string");
```

QUESTION A3**[24 marks]**

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

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

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

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

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

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

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

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

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

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

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

FIRST(Entry) { wordSym }

FOLLOW(Entry) { EOFSym, wordSym }

FIRST(References) { appendixSym, dashDashSym, numberSym }

FOLLOW(References) (EOLSym)

FIRST(PageRefs) { appendixSym, numberSym }

FOLLOW(PageRefs) { commaSym, EOLSym }

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

```
static void Index() {
    // Index = { Entry } EOF .
    while (sym == wordSym) {
        Entry();
    }
    Accept(EOLSym, "EOF expected");
} // Index

static void Entry() {
    // Entry = Key References EOL .
    Key(); References();
    Accept(EOLSym, "entries must be terminated by end-of-line");
} // Entry

static void Key() {
    // Key = word { ", " word | word } .
    Accept(wordSym, "word expected");
    while (sym == commaSym || sym == wordSym) {
        if (sym == commaSym) GetSym();
        Accept(wordSym, "word expected");
    }
} // Key

static void References() {
    // References = DirectRefs | CrossRef .
    switch(sym) {
        case appendixSym:
        case numberSym:
            DirectRefs(); break;
        case dashDashSym:
            CrossRef(); break;
        default: // this one tends to get "forgotten"
            Abort("invalid reference");
    }
} // References

static DirectRefs() {
    // DirectRefs = PageRefs { ", " PageRefs } .
    PageRefs();
    while (sym == commaSym) {
        GetSym(); PageRefs();
    }
} // DirectRefs
```

```

static void PageRefs() {
// PageRefs = [ "Appendix" number ",," ] number [ "-" number ] .
if (sym == appendixSym) {
    GetSym();
    Accept(numberSym, "number expected");
    Accept(commaSym, "comma expected");
}
Accept(numberSym, "number expected");
if (sym == dashSym) {
    GetSym();
    Accept(numberSym, "number expected");
}
} // PageRefs

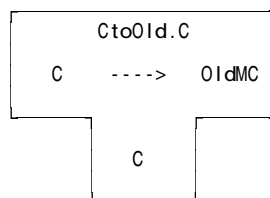
static CrossRef() {
// CrossRef = "---" "see" Key .
Accept(dashDashSym, "-- expected");
Accept(seeSym, "see expected");
Key();
} // CrossRefs

```

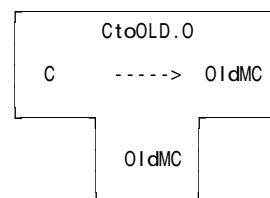
QUESTION A4**[4 + 4 + 3 + 3 = 14 marks]**

(T diagrams) The process of "porting" a compiler to a new computer incorporates a *retargetting* phase (modifying the compiler to produce target code for the new machine) and a *rehosting* phase (modifying the compiler to run on the new machine). Illustrate these two phases for porting a C compiler, by drawing a set of T diagrams. Assume that you have available the compilers (a) and (b) below and wish to produce compiler (c). [16 marks]

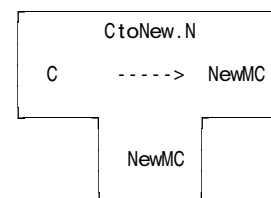
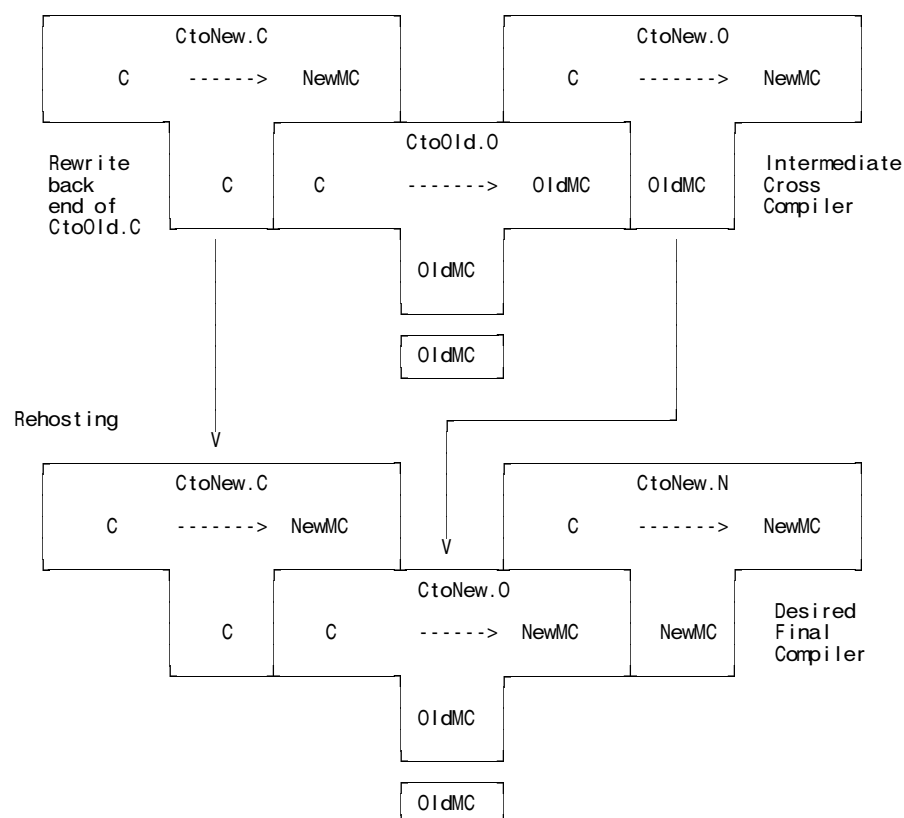
(a) Old Compiler Source



(b) Old Compiler Executable

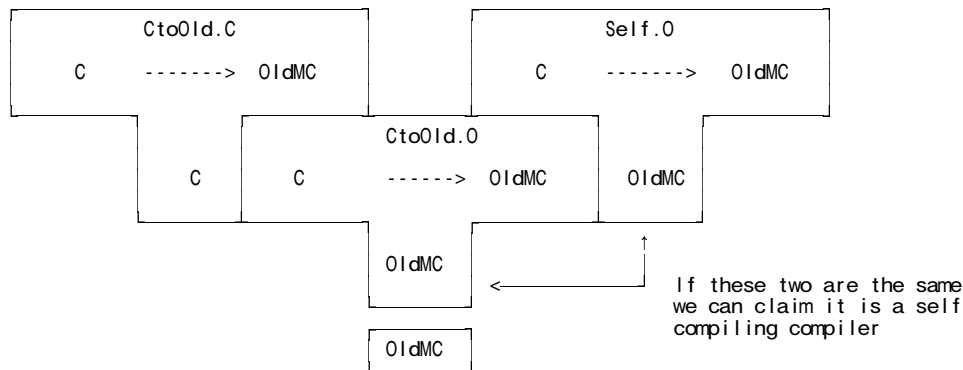


(c) New Compiler Executable

**Retargetting**

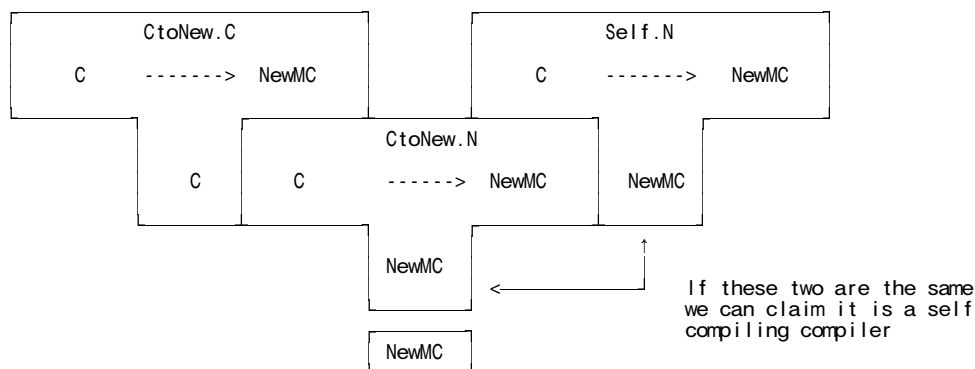
Check for being a self compiling compiler.

It should be. To test, try compiling the source of the old compiler with the executable supplied for OldMC



Is the new compiler also a self-compiling compiler?

It should be. To test, try compiling the source of the new compiler with the executable produced in parts (a) and (b)



QUESTION A5

[16 marks]

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

```
<!-- comment - a sample extract from an XML file -->
<personnel>
  <entry>
    <name>John Smith</name>
  </entry>
  <entry_2>
    <name>Joan Smith</name>
    <address/>
    <gender>female</gender>
  </entry_2>
</personnel>
```

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

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

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

and give PRODUCTIONS that describe complete documents like the one illustrated. You may do this conveniently on the page supplied at the end of the examination paper.

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

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

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

This problem is totally unseen, but straightforward

```
COMPILER XML $CN
/* Parse a set of simple XML elements (no attributes)
   P.D. Terry, Rhodes University, 2017 */

CHARACTERS
  letter   = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .
  lowline  = "_" .
#  intag   = letter + "0123456789_." .
#  intext  = ANY - "<" .
#  incoment = ANY - "-" .

TOKENS
#  opentag = "<" ( letter | lowline ) { intag } ">" .
#  closetag = "</" ( letter | lowline ) { intag } ">" .
#  emptytag = "<" ( letter | lowline ) { intag } "/>" .
#  text    = intext { intext } .

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

IGNORE CHR(0) .. CHR(31)

PRODUCTIONS
  XML = Element { Element } .

#  Element =
#          opentag      ( . String open; . )
#          { Element
#            | text
#          }
#          closetag      ( . if (!token.val.Substring(2).Equals(open))
#                        SemError("mismatched tag"); . )
#          | emptytag .

END XML.
```

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

Section B [90 marks]

Please note that there is no obligation to produce a machine readable solution for this section. Coco/R and other files are provided so that you can enhance, refine, or test your solution if you desire. If you choose to produce a machine readable solution, you should create a working directory, unpack EXAM.ZIP, modify any files that you like, and then copy all the files back onto an exam folder on the network.

QUESTION B6**[4 marks]**

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

This is standard stuff. Syntax is concerned (usually) with context-free representation of statements and programs, so that

```
A = B + C;
```

is a syntactically correct assignment, whereas

```
A = B + C +;
```

is not. Static semantics refers to those context-sensitive features that can be checked at compile time without actually "executing" the program. So a set of statements like

```
int A;
string B;
bool C;
A = B + C;
```

while all being syntactically acceptable, would (probably) be meaningless, as addition of two disparate values followed by assignment to an even more disparate target could have no sensible meaning.

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

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

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

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

Yes of course we can. A constant is just the simplest form of expression.

Perhaps the second part is a trick question. The grammar they have been given does not define any other constants, except string constants. I suspect to find answers that mention integer constants like 1234, rather than strings, which tend to be forgotten in Parva. A later question might lead to the correct answer that the Boolean constants 0 and 1 could be made permissible (see question B13).

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

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

or

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

Any student who gets this wrong does not deserve to have made it through to third year. Yes, both calls are valid. It is the value of an expression that is being passed, not the address of one, and an argument specified by a single variable is a very simple form of expression.

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

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

This requires a bit more work. The symbol table entry for a function needs to record the number of formal parameters for each function, and this needs to be checked when we parse an argument list.

```

FunDefinition                                     (. string name;
#                                                     int nParams; .)
# = FunName<out name>                             (. varTable = new VarTable(); .)
#   "(" ParamList<out nParams> WEAK ")"           (. Label entryPoint = new Label(known);
#                                                     FunTable.AddEntry(new FunEntry(name,
#                                                         nParams,
#                                                         entryPoint)); .)

    "returns" Definition                         (. CodeGen.FunctionTrap(); .) .

# ParamList<out int nParams>                       (. char name;
#                                                     nParams = 0; .)
# = [ Variable<out name>                           (. varTable.Add(name); nParams++; .)
#   { WEAK " ," Variable<out name>                 (. varTable.Add(name); nParams++; .)
#   }
# ] .

FunctionCall                                     (. String name; .)
# = FunName<out name>                             (. FunEntry entry = FunTable.findEntry(name);
#                                                     if (!entry.defined)
#                                                         SemError("unknown function");
#                                                     CodeGen.frameHeader(); .)

#   "(" ArgumentList<entry.nParams>
#       WEAK ")"
#                                                     (. CodeGen.call(entry.entryPoint); .) .

# ArgumentList<int args>
# = [ Expression
#   { WEAK " ," Expression
#   } ]
#   (. args--;
#   (. args--;
#   (. if (args != 0)
#       SemError("formal/actual parameter mismatch"); .)

```

Another solution that did not occur to me at first, but might be suggested by candidates is:

```

FunDefinition                                     (. string name; .)
# = FunName<out name>                             (. varTable = new VarTable(); .)
#   "(" ParamList WEAK ")"                       (. Label entryPoint = new Label(known);
#                                                     FunTable.AddEntry(new FunEntry(name,
#                                                         varTable.varList.Count,
#                                                         entryPoint)); .)

    "returns" Definition                         (. CodeGen.FunctionTrap(); .) .

```

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

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

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

If Fun(x, y) is called with two few arguments, the effect is to OR two values, the first of which will be the value of the argument x, and the second of which will be whatever value happens to be in the stack element that should have been initialised to the value of the missing second argument. So the effect is to produce a wrong, generally unpredictable answer. If Fun (x,y) is called with too many arguments, the extra ones will be pushed onto the stack in positions that the function will never address, and so the effect will be to return the OR of the first two arguments only.

I thought at first that the stack pointer would be corrupted. In fact this does not happen, because of the way in which the CALL and RET codes are interpreted. The stack and frame pointer are saved in the frame header, and restored from this later. If they had been manipulated using the count of the actual arguments rather than the count of the formal arguments, trouble would have soon followed. So the design of the PVM has been safe after all, but it is still sensible to modify the system to compare the counts.

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

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

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

Justify your answer. [2 marks]

There is no problem at all - just as any program might declare variables or define methods that are never used, no harm is done (other than wasting a bit of space).

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

This can be done with a slight hack to the symbol table, as below. It cannot be done by simply counting the number of times the Find method is called, as some candidates might try:

```
class Entry {
    public char name;
    public int  offset;
    public int  level = local;
#    public bool used = false;

    public Entry(char name, int offset) {
        this.name  = name;
        this.offset = offset;
#        this.used  = false;

        } // constructor

    public override string ToString() {
#        return name + " " + offset + " " + used;
        } // ToString

    } // Entry

class VarTable {
    // Symbol tables for single letter variables and parameters

    List<Entry> varList = new List<Entry>();

    Entry sentinel = new Entry('?', 0);

    public Entry Find(char name) {
        // Searches table for an entry matching name.
        // If found then returns the corresponding entry
        for (int look = 0; look < varList.Count; look++)
            if (varList[look].name == name) {
#                varList[look].used = true;
                return varList[look];
            }
        Parser.SemError("undeclared");
        return sentinel;
    } // VarTable.Find

#    public void CheckUnused() {
#        // Checks for unreferenced parameters
#        for (int i = 0; i < varList.Count; i++)
#            if (!varList[i].used)
#                Parser.Warning("parameter " + varList[i].name + " never referenced");
#    } // VarTable.CheckUnused

    } // end VarTable
```

and to the grammar

There are several other ways of doing it, for example, when parsing a function call to check that the name of the function being called is not the same as the name of the function being defined:

QUESTION B12**[8 marks]**

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

This is very easy if you are really familiar with the Parva compiler.

```

Expression
# = AndExp                                (. Label shortCircuit = new Label(!known); .)
# { ( "or" | "+" )                        (. CodeGen.BooleanOp(shortCircuit, CodeGen.or); .)
#   AndExp
# }                                         (. shortCircuit.Here(); .)

AndExp
# = EqExp                                (. Label shortCircuit = new Label(!known); .)
# { [ "and" | "." ]                      (. CodeGen.BooleanOp(shortCircuit, CodeGen.and); .)
#   EqExp
# }                                         (. shortCircuit.Here(); .)

```

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

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

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

The modifications for *and* and *or* are included in the solution to Question B12. The remaining changes are

```

NotExp
# = Factor { ""                            (. CodeGen.NegateBoolean(); .)
#   | "not" Factor                        (. CodeGen.NegateBoolean(); .)

Constant<out int value>
# = ( "true" | "1" )                      (. value = TRUE; .)
#   | ( "false" | "0" )                  (. value = FALSE; .)

```

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

```

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

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

```

| x | y | x.y | x | y | x.y |
|---|---|-----|-------|-------|-------|
| 0 | 0 | 0 | false | false | false |
| 0 | 1 | 0 | false | true | false |
| 1 | 0 | 0 | true | false | false |
| 1 | 1 | 1 | true | true | true |

We add to the PRAGMA section

```

PRAGMAS
...
# NumericOn = "$N+" .                (. numeric = true; .)
# NumericOff = "$N-" .                (. numeric = false; .)

```

and then modify the WriteElement actions:

```

WriteElement                                     (. string str; .)
= StringConst<out str>                         (. CodeGen.WriteString(str); .)
# | Expression                                (. if (numeric) CodeGen.Write(Types.intType);
#                                         else CodeGen.Write(Types.boolType); .) .

```

QUESTION B14**[8 marks]**

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

```

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

```

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

This can be done by adding another field mutable (changeable) to an entry in the varTable.

The field is set true when the table entries are first made. However, when a variable is identified as a loop control, the value of its mutable field is saved, and then set to false. The sections in the grammar where a variable might be altered then incorporate a test of this field, which is restored again at the end of generating code for the loop.

```

LoopStatement                                     (. char name; .)
= "loop"                                         (. Entry var = varTable.Find(name);
# Variable<out name>                             if (!var.mutable)
#                                             SemError("loop variable may not be altered");
#                                             bool mutable = var.mutable; // save
#                                             var.mutable = false; // protect
#                                             (. var.mutable = mutable; /* restore */ .) .
# Statement

```

```

Assignment                                     (. char name; .)
= Variable<out name>                             (. Entry var = varTable.Find(name);
#                                             if (!var.mutable)
#                                             SemError("loop variable may not be altered"); .)
# "=" Expression
# WEAK ";" .
#                                             (. CodeGen.StoreValue(var); .)

```

```

ReadElement                                     (. string str;
# char name; .)
# = StringConst<out str>
# | Variable<out name>
#                                         (. CodeGen.WriteString(str); .)
#                                         (. Entry var = varTable.Find(name);
#                                         if (!var.mutable)
#                                         SemError("loop variable may not be altered");
#                                         CodeGen.LoadAddress(var);
#                                         CodeGen.Read(Types.boolType); .) .
#

```

QUESTION B15**[8 marks]**

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

```

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

```

as equivalent to

```

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

```

and that the code generated follows that template, leading to

```

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

```

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

There are probably several ways of cutting down on the amount of generated code. Here is one of them

```

                LDC  0
                STL  x      ; x = false;
start          .....      ; code for the body of the loop goes here
                LDL  x      ; push current value of x to await later test
                LDC  1
                STL  x      ; x = true; ready for next iteration of loop
                BZE  start   ; retrieve previous x from TOS for this test
exit           ; rest of code after loop

```

for which the code generation could be accomplished as follows:

| | |
|--------------------|---|
| LoopStatement | (. char name; .) |
| = "loop" | |
| Variable<out name> | (. Entry var = varTable.Find(name); |
| | if (!var.mutable) |
| | SemError("loop variable may not be altered"); |
| | CodeGen.LoadConstant(0); |
| | CodeGen.StoreValue(var); |
| | bool mutable = var.mutable; // save |
| | var.mutable = false; |
| | Label startLoop = new Label(known); .) |
| # Statement | (. CodeGen.LoadValue(var); |
| # | CodeGen.LoadConstant(TRUE); |
| # | CodeGen.StoreValue(var); |
| # | CodeGen.BranchFalse(startLoop); |
| | var.mutable = mutable; /* restore */ .) . |

Section C

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

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

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

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

Section D

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

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