

# RHODES UNIVERSITY

## November Examinations - 2012

### Computer Science 301 - Paper 2 - Solutions

Examiners:

Prof P.D. Terry  
Prof D.G. Kourie

Time 4 hours

Marks 180

Pages 16 (please check!)

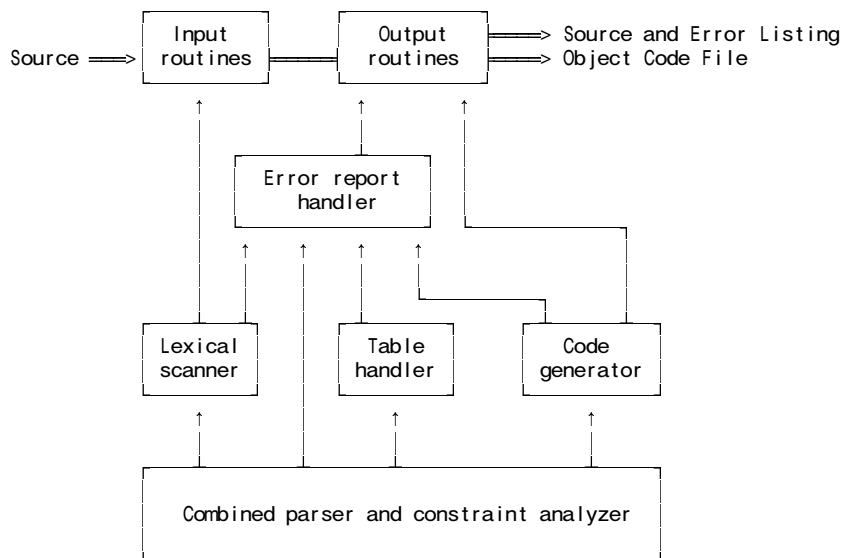
**Answer all questions. Answers may be written in any medium except red ink.**

(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 calculator based on the Parva Virtual Machine interpreter system studied in the course. Some 16 hours before the examination a complete grammar for a calculator 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 [ 100 marks ]

- A1. (Compiler structure) A syntax-directed compiler usually incorporates various components, of which the most important are the scanner, parser, constraint analyser, error 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. Also indicate which components constitute the *front end* and which the *back end* of the compiler. [10 marks]

The diagram they were given in the text looks like this:



The back end incorporates the code generator and some output routines. The rest is essentially "front end".

- A2. (Recursive descent parsers) The following Cocol grammar describes the form of an index to a textbook and should be familiar from the practical course.

```
COMPILER Index $CN
/* Grammar describing index in a book
   P.D. Terry, Rhodes University, 2012 */

CHARACTERS
/* Notice the careful and unusual choice of character sets */
digit    = "0123456789" .
startword = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" + "'" .
inword    = startword + digit + "-+" .
eol       = CHR(10) .
```

```

TOKENS
/* Notice the careful and unusual definition for word */
word      = startword { inword } .
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 ] .
CrossRef    = "---" "see" Key .
END Index .

```

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 enumeration

```

EOFSym, noSym, EOLSym, wordSym, numberSym, appendixSym, commaSym,
dashSym, dashDashSym, seeSym

```

Develop a hand-crafted recursive descent parser for recognizing the index of a book based on the grammar above. (*Your parser can take drastic action if an error is detected. Simply call methods like `accept` and `abort` 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.*) [16 marks]

```

static void Index() {
    // Index = { Entry } EOF .
    while (sym == wordSym) {
        Entry();
    }
    accept(EOFSym, "EOF expected");
}

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

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

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");
    }
}

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

```

```

static PageRefs() {
// PageRefs = [ "Appendix" ] number [ "-" number ] .
  if (sym == appendixSym) getSym();
  accept(numberSym, "number expected");
  if (sym == dashSym) {
    getSym();
    accept(numberSym, "number expected");
  }
}

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

```

A3. (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  $\alpha$  and  $\beta$ , 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]

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

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

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

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

$$(3) \text{ A context-free grammar}$$

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

$$(4) \text{ A reduced grammar}$$

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  $\alpha$  and  $\beta$ , and where

$$B \Rightarrow^* \gamma$$

for some  $\gamma \in T^*$ .

- (b) In terms of the notation here, concisely state the two 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. [6 marks]

**Rule 1**

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  $\xi_{ik}$  must be disjoint, that is

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

**Rule 2**

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

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

but where  $\xi_{ik} \Rightarrow \varepsilon$  for some  $k$ , the sets of initial terminal symbols of all sentences that can be generated from each of the  $\xi_{ij}$  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,

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

- (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. Give such an algorithm, using a Java-like notation. [3 marks]

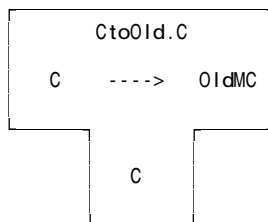
```

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");

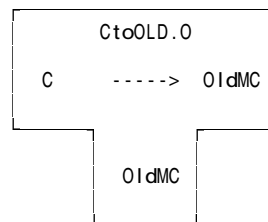
```

- A4. (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]

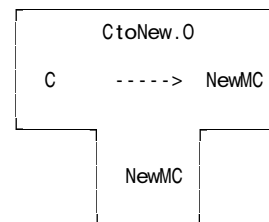
(a) Old Compiler Source



(b) Old Compiler Executable

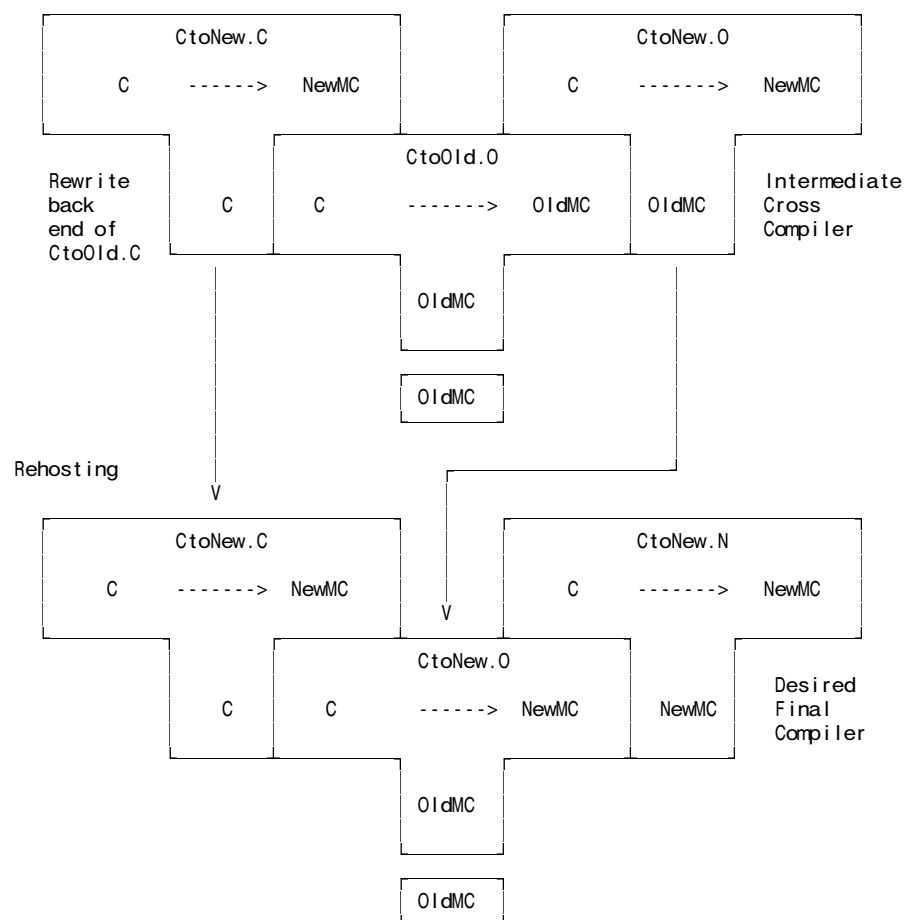


(c) New Compiler Executable



(You may conveniently make use of the outline T-diagrams at the end of this paper; complete these and attach the page to your answer book.)

#### Retargetting



- A5. (*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. [18 marks]

*This problem is totally unseen. Here is a C# solution. The Java one is virtually identical:*

```
COMPILER XML $CN
/* Parse a set of simple XML elements (no attributes)
   P.D. Terry, Rhodes University, 2012 */

CHARACTERS
  letter   = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .
  lowline  = "_" .
  intag    = letter + "0123456789_.-" .
  inword   = ANY - "<" .
  incomment = ANY - "--" .

TOKENS
  opentag  = "<" ( letter | lowline ) { intag } ">" .
  closetag = "</" ( letter | lowline ) { intag } ">" .
  emptytag = "<" ( letter | lowline ) { intag } "/>" .
  word     = inword { inword } .

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

```

IGNORE CHR(0) .. CHR(31)

PRODUCTIONS
XML      = Element { Element } .

Element =
    opentag      ( . String open; . )
    { Element   ( . open = token.val.Substring(1); . )
    | word
    | emptytag
    }
    closetag     ( . if (!token.val.Substring(2).Equals(open))
                  SemErr("mismatched tag"); . )
    .

END XML .

```

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

- A6. (*Code generation*) A BYT (Bright Young Thing) has been nibbling away at writing extensions to her first Parva compiler. It has been suggested that a function that will return the maximum element from a variable number of arguments would be a desirable addition, one that might form part of an expression:

$$a = \max(x, y, z) + 5 - \max(a, \max(c, d));$$

and that this could be achieved by extending the production for a *Factor* in a fairly obvious way, and adding a suitable opcode to the PVM. To refresh your memory, the production for *Factor* in the simple Parva compiler is defined as follows:

```

Factor<out int type>      ( . int value = 0;
                          type = Entry.noType;
                          int size;
                          DesType des;
                          ConstRec con; . )
= Designator<out des>    ( . type = des.type;
                          switch (des.entry.kind) {
                              case Entry.Var:
                                  CodeGen.dereference();
                                  break;
                              case Entry.Con:
                                  CodeGen.loadConstant(des.entry.value);
                                  break;
                              default:
                                  SemError("wrong kind of identifier");
                                  break;
                          } . )
| Constant<out con>      ( . type = con.type;
                          CodeGen.loadConstant(con.value); . )
| "new" BasicType<out type>
  "[" Expression<out size> ( . type++; . )
  "[" Expression<out size> ( . if (!isArith(size))
                          SemError("array size must be integer");
                          CodeGen.allocate(); . )
  "]"
| "!" Factor<out type>    ( . if (!isBool(type)) SemError("boolean operand needed");
                          else CodeGen.negateBoolean();
                          type = Entry.boolType; . )
| "(" Expression<out type> ")"
.

```

while a sample of the opcodes in the PVM that deal with simple arithmetic and logic arithmetic were interpreted with code of the form

```

case PVM ldc:           // push constant value
    push(next());
    break;
case PVM add:           // integer addition
    tos = pop(); push(pop() + tos);
    break;
case PVM sub:           // integer subtraction
    tos = pop(); push(pop() - tos);
    break;
case PVM not:           // logical negation
    push(pop() == 0 ? 1 : 0);
    break;

```

Suggest, in as much detail as time will allow, how the *Factor* production and the interpreter would need to be changed to support this language extension.

Allow your *max* function to take one or more arguments, and ensure that it can only be applied to arithmetic arguments. Assume that a suitable *CodeGen* routine can be introduced to generate any new opcodes required. [16 marks]

*This problem is totally unseen. The addition to the Factor (or Primary, if one wishes to use the grammar in the exam kit) parser would consist of another case arm. Here are two possibilities:*

```
| "max" "(" Expression<out type>  (. if (!isArith(type))
                                SemError("argument must be numeric"); .)
  { "," Expression<out type>      (. if (!isArith(type))
                                SemError("argument must be numeric");
                                CodeGen.max(); .)
  } ")"
```

```
| "MAX" "(" Expression<out type>  (. int count = 1;
                                if (!isArith(type))
                                    SemError("argument must be numeric"); .)
  { "," Expression<out type>      (. count++;
                                if (!isArith(type))
                                    SemError("argument must be numeric"); .)
  } ")"                          (. CodeGen.max2(count); .)
```

*In the second case, the new opcode MAX2 N has the number of expressions that have been stacked up as its argument N. Finding the maximum then can be done by popping pairs of values and pushing back the larger. In terms of operations suggested for the opcodes given earlier this might be achieved as follows*

```
case PVM.max:                // max(tos, sos)
    tos = pop();
    sos = pop();
    if (tos > sos) push(tos); else push(sos);
    break;

case PVM.max2:                // max(a,b,c,...)
    int count = next();
    while (count > 1) {
        tos = pop();
        sos = pop();
        if (tos > sos) push(tos); else push(sos);
        count--;
    }
    break;
```

*but of course other code is possible (and more efficient) if it manipulates *cpu.sp* directly.*

## Section B [ 80 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 B7

[ 4 marks ]

After studying the *CalcPVM* 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 B8

[ 6 marks ]

The grammar makes no provision for checking that an expression on the right hand side of an assignment must be of integer type, that the value returned by a function must be of integer type, and that the arguments used in a function call must be of integer type. Modify it so that these checks will be carried out.

<pre>Assignment #   = Variable&lt;out name&gt; #   "=" Expression&lt;out expType&gt; # #   WEAK ";" .</pre>	<pre>(. char name;   int offset;   int expType; .) (. offset = varTable.FindOffset(name); .) (. if (!IsArith(expType))   SemError("integer expression expected");   CodeGen.StoreValue(offset); .)</pre>
<pre>Definition #   = Expression&lt;out expType&gt; # #   WEAK ";" .</pre>	<pre>(. int expType; .) (. if (!IsArith(expType))   SemError("a function must return an integer value");   CodeGen.StoreReturnValue(expType);   CodeGen.LeaveFunction(); .)</pre>
<pre>ArgumentList #   = [ Expression&lt;out expType&gt; #     { WEAK "," Expression&lt;out expType&gt; #     } ] .</pre>	<pre>(. int expType; .) (. if (!IsArith(expType))   SemError("arguments must be of integer type"); .) (. if (!IsArith(expType))   SemError("arguments must be of integer type"); .)</pre>

### QUESTION B9

[ 3 + 3 = 6 marks ]

- (a) Can you use constants as arguments in a function call - for example

```
Fun(x, y) returns x + y;
write(Fun(200, 400));
```

Why? And if so, might there be any constants that you could not use in this way? Explain.

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

Perhaps the second part is a trick question. We could not use the constants **true** and **false** as they are deemed to be of Boolean type!

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

```
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 B10****[ 8 + 4 = 12 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.

*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, localOffset); nParams++; .)
#     { WEAK " ," Variable<out name>       (. varTable.Add(name, localOffset); nParams++; .)
#     }
#   ] .

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

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

ArgumentList<int args>                      (. int expType; .)
#   = [ Expression<out expType>            (. args--;
#                                         if (!isArith(expType))
#                                           SemError("arguments must be of integer type"); .)

#   { WEAK " ," Expression<out expType>    (. args--;
#                                         if (!isArith(expType))
#                                           SemError("arguments must be of integer type"); .)

#   } ]                                     (. if (args != 0)
#                                         SemError("numbers of formal and actual parameters do not match"); .)
#

```

*Another solution that had not occurred to me, but was suggested correctly by several 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.size(),
#                                         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 a program of the form

```

Fun(x, y) returns x + y;    // two parameters
...
a = Fun(x) + Fun(x, y, z);  // one and then three arguments

```

*If Fun(x, y) is called with two few arguments, the effect is to add two numbers, 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 sum 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 B11

[ 3 + 8 = 11 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.

*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 alert a user to this situation, show the changes to the code needed to check for it.

*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 FindOffset method is called, as some candidates tried:*

```
class VarEntry {
    public char name;
    public int offset;
#    public bool used;

    public VarEntry(char name, int offset) {
        this.name = name;
        this.offset = offset;
#        this.used = false;
    } // constructor

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

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

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

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

#    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");
#    } // CheckUnused
} // end VarTable
```

and to the grammar

```

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

```

### QUESTION B12

[ 4 + 4 = 8 marks ]

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

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

and

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

The first of these require no additions - the reference to parameter *z* will simply show up as undeclared in any case, as the code for this check was provided in the supplied table handler class.

The second one requires that the `Add` method requires a preliminary check for a repeated parameter name:

```

public void AddVar(char name, int headerSize) {
  // Adds an entry, computing what its offset would be in a stack frame
#   for (int look = 0; look < varList.Count; look++)
#     if (varList[look].name == name) {
#       Parser.SemError("redeclared");
#       return;
#     }
  varList.Add(new VarEntry(name, varList.Count + headerSize));
} // AddVar

```

### QUESTION B13

[ 3 + 3 + 3 = 9 marks ]

(a) Under what conditions can one function call another - in other words, can one write code like

```

Double(x)  returns 2 * x;
Triple(x)  returns x + Double(x);

```

Yes, one can write code like this provided that a function refers only to functions that have already been defined.

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

```

Silly(x)  returns 2 * Silly(x);
write ( Silly(x) );

```

The system will enter an infinite loop and eventually collapse with a stack overflow.

(c) What modification to the supplied system will prevent that Silly sort of behaviour? Explain.

One can do this by delaying the addition to the Function Table until after the Definition has been parsed:

```

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

```

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 B14****[ 14 marks ]**

You may have wondered why it was suggested that you leave all the machinery for evaluating Boolean expressions in place when the whole system seems geared to integers only. But consider - if we redefine the form of a function specification to be

```
FunDefinition
= "(" ParamList ")"
  "returns" Definition .

Definition
= Expression ";"
  | "if" "(" Expression ")" [ "returns" ] Definition "else" [ "returns" ] Definition .
```

then we should presumably be allowed to write a recursive definition like

```
Factorial(n) returns if (n <= 0) returns 1; else returns n * Factorial(n - 1);
```

Add the necessary extensions to the system to incorporate this powerful feature.

*Well, you can't have it both ways. If you want to be able to write recursive definitions you will have to leave well alone and enter the function into the table as soon as you have parsed the parameter list:*

```
FunDefinition                                (. string name;
= FunName<out name>                          int nParams; .)
"(" ParamList<out nParams> WEAK ")"          (. varTable = new VarTable(); .)
"returns" Definition                        (. Label entryPoint = new Label(known);
                                           FunTable.AddEntry(new FunEntry(name, nParams, entryPoint)); .)
                                           (. CodeGen.FunctionTrap();
                                           varTable.CheckUnused(); .) .
```

*The code for Definition now has to handle code generation for an if-else construct similar to that used in the Parva compiler, save that the else clause is mandatory. All very easy, really!*

```
Definition                                (. int expType; .)
= Expression<out expType>                  (. if (!IsArith(expType))
                                           SemError("a function must return an integer value");
                                           CodeGen.StoreReturnValue(expType);
                                           CodeGen.LeaveFunction(); .)

# WEAK ";"
# | "if" "(" Expression<out expType>        (. if (!IsBool(expType))
#                                           SemError("boolean expression needed"); .)
#                                           (. Label falseLabel = new Label(!known);
#                                           CodeGen.BranchFalse(falseLabel); .)
#                                           (. falseLabel.Here(); .)
# ")"
# [ "returns" ] Definition
# "else" [ "returns" ] Definition .
```

**QUESTION B15****[ 4 + 6 = 10 marks ]**

(a) What would be the effect of replacing the syntax suggested in question B14 with

```
FunDefinition
= "(" ParamList ")"
  "returns" Definition .

Definition
= Expression ";"
  | "if" "(" Expression ")" Expression ";" "else" Expression ";" .
```

*You would not be able to write nested if-then-else structures - so could not handle a Fibonacci function, for example:*

```
Fib(n) returns if (n == 0) 0;
               else if (n == 1) 1;
               else (Fib(n-2) + Fib(n-1));
```

- (b) What would be the effect of replacing the syntax suggested in question B14 by

```
FunDefinition
= "(" ParamList ")"
  "returns" Definition .

Definition
= Expression ";"
  | "if" "(" Expression ")" Definition [ "else" Definition ] .
```

*I was looking for two insights here. One would have the "dangling else" phenomenon to contend with, though code for that is easily written. More seriously, omission of the else clause would result in a function that potentially did not return a value at all. The supplied system makes use of a TRAP opcode to detect this at run-time, but one shudders to think how much code has been written in some languages whose implementations do not raise some sort of error or exception if this happens, and simply return complete rubbish ...*

*For the record, if one were silly enough to make the else clause optional the code generation would look something like*

<pre>Definition = Expression&lt;out expType&gt;  #      ";" #        "if" "(" Expression&lt;out expType&gt; #      #      ")" #      [ "returns" ] Definition #      ( "else" [ "returns" ] #      #      Definition #        /* no else part */ #      ) .</pre>	<pre>(. int expType; .) (. if (!IsArith(expType))   SemError("a function must return an integer value");   CodeGen.StoreReturnValue(expType);   CodeGen.LeaveFunction(); .)  (. if (!IsBool(expType))   SemError("boolean expression needed"); .) (. Label falseLabel = new Label(!known);   CodeGen.BranchFalse(falseLabel); .)  (. Label outLabel = new Label(!known);   CodeGen.Branch(outLabel);   falseLabel.Here(); .) (. outLabel.Here(); .) (. falseLabel.Here(); .)</pre>
---	--

## 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 extend a version of the supplied calculator grammar so as to define simple "one-liner" functions that could be incorporated into the evaluation of 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. Although there had been some discussion of the mechanisms in lectures, there had been no room to explore these in the practical course. However, a CFG for the Parva language (but devoid of attributes and actions) had been extended in an earlier practical, so candidates who had completed that exercise should surely have seen the connection. A skeleton symbol table handler was provided, as was a code generator virtually identical to the one they had seen previously.

They were provided with an exam kit for Java or 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 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 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.

## **Free information**

Listings were provided of the following:

### **Summary of useful library classes**

#### **Strings and Characters in Java**

#### **Simple list handling in Java**

#### **Simple list handling in C#**