RHODES UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE

EXAMINATIONS: JANUARY/FEBRUARY 2017

Computer Science 301

PAPER 1 - Translators - Solutions

Internal Examiner: Prof P.D. Terry

MARKS: 160
DURATION: 4 hours

External Examiner: Prof M. Kuttel

GENERAL INSTRUCTIONS TO CANDIDATES

---

1. This paper consists of 16 pages and 29 questions. Please ensure that you have a complete paper.
2. Answers must be written on the question paper, and may be written in any medium except red ink.
3. You may attempt all questions in Section A, and then choose to answer either Section B or Section C.

Student Number

Seat Number

Most of these questions can and should be answered by marking the CORRECT option or options offered as possible solutions. Use either a tick or a cross. Do NOT use a tick to mean yes and cross to mean no!

To discourage guessing, candidates are warned that incorrect answers may carry a negative penalty. If you do not know the answer, it may be better to mark the "don't know" box.

The maximum score that can be earned on this paper is 160. However, a mark of 140 will be taken as worth 100%.

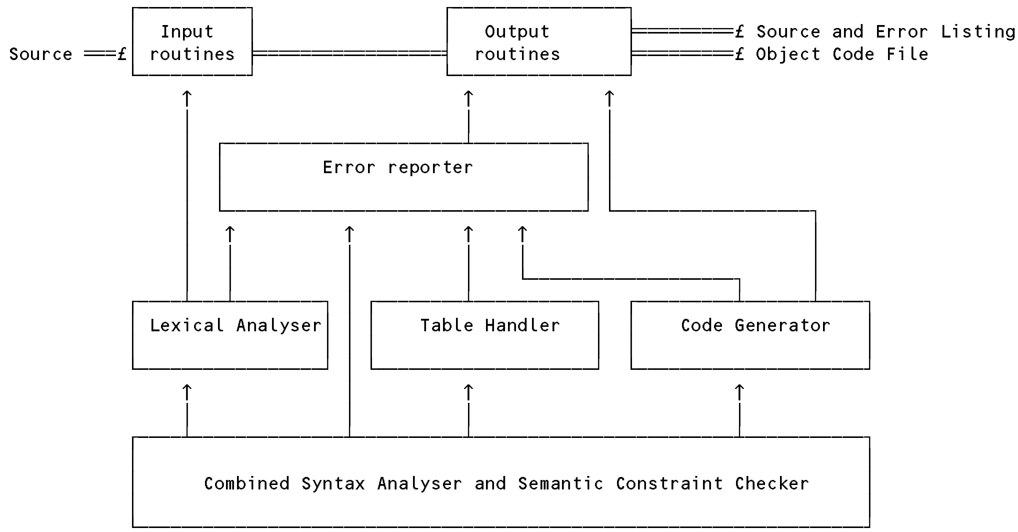
Your "exam kit" contains the source code needed to build a Parva compiler like the one that would have formed a complete solution to the November examinations, with the extensions suggested to you during the "24 hour" preparation period. There is a precompiled version of this in the file PREPARVA.EXE which you are free to use in any way you see fit.

PLEASE DO NOT TURN OVER THIS PAGE UNTIL TOLD TO DO SO

SECTION A - 25 questions in all

Question A1 [6 marks]

A compiler conventionally has six components (a) Lexical analyser (b) Syntax analyser (c) Semantic constraint analyser (d) Code generator (e) Symbol table handler (f) Error reporter. Use the diagram below to indicate where each of these components can be located in a way that shows their interdependence.



Question A2 [6 marks]

For each of the components of a compiler mentioned in question 1, indicate whether it would be considered to be associated with the front end, the back end, or both ends of the compiler.

Lexical analyser	Front *	Back	Both	Don't know
Syntax analyser	Front *	Back	Both	Don't know
Semantic constraint analyser	Front *	Back	Both	Don't know
Code generator	Front	Back *	Both	Don't know
Symbol table handler	Front *	Back	Both	Don't know
Error reporter	Front	Back	Both *	Don't know

Question A3 [3 marks]

Components of a compiler may be called by several names. Complete the table below.

A lexical analyser is sometimes called a	Scanner *	Parser	Don't know
A syntax analyser is sometimes called a	Scanner	Parser *	Don't know
White space and comments are ignored by the	Scanner *	Parser	Don't know

Question A4 [4 marks]

What two pieces of software must you have available to you so that you can claim that you possess a self-compiling compiler for language X?

Native code executable version of a compiler for language X	*
Native code executable version of a compiler for language C#	
Source code of the executable version of compiler for X, written in X	*
Source code of the executable version of compiler for X, written in C#	
A Cocol grammar for the compiler for X, expressed as an ATG file	
An executable Coco/R compiler	

Question A5 [4 marks]

Consider the following compilers known to you. Which of the following claims is true for each one? (More than one claim may be true.)

The Coco/R compiler generator COCOR.EXE is

a high level compiler *	a native code compiler	a self-compiling compiler *	an interpretive compiler

The .NET C# compiler CSC.EXE is

a high level compiler	a native code compiler *	a self-compiling compiler	an interpretive compiler

The Parva compiler PARVA.EXE generated by using COCOR.EXE and CSC.EXE is

a high level compiler	a native code compiler	a self-compiling compiler	an interpretive compiler *

Question A6 [4 marks]

Consider the following C# statement. How many language tokens appear in this statement altogether? (Hint: in the statement `a = ( a + ( a ) ) ;` there are 10 tokens altogether but only 6 different ones).

```
while ( i <= 10 && !list[i].name.Equals("Pat Terry")) i++;
```

22	tokens

Question A7 [4 marks]

If we define a language L by a grammar G, we speak of G = G(N, T, S, P) as having four components, conventionally called its N (non-terminals), T (terminals), S (start) and P (productions). Which of these are sets?

N *	T *	S	P *	None	Don't know

Question A8 [4 marks]

Here are three sets of productions from which algebraic-like strings containing - and * operators and *a, b* and *c* operands may be derived - for example *a * b * a*.

(G1)	*Expression*	=	*Term* | *Expression* "-" *Term* .	(1, 2)
	Term	=	*Factor* | *Term* "*" *Factor* .	(3, 4)
	Factor	=	"*a*" | "*b*" | "*c*" .	(5, 6, 7)

(G2)	*Expression*	=	*Term* | *Term* "-" *Expression* .	(1 ,2)
	Term	=	*Factor* | *Factor* "*" *Term* .	(3, 4)
	Factor	=	"*a*" | "*b*" | "*c*" .	(5, 6, 7)

(G3)	*Expression*	=	*Term* | *Term* "*" *Expression* .	(1, 2)
	Term	=	*Factor* | *Factor* "-" *Term* .	(3, 4)
	Factor	=	"*a*" | "*b*" | "*c*" .	(5, 6, 7)

Which, if any, of these grammars describes strings in which the operators and operands obey the usual rules of associativity, distributivity and precedence?

None	G1 *	G2	G3	G1 and G2	G2 and G3	G3 and G1	G1 and G2 and G3	Don't know

Question A9 [8 marks]

The following represent attempts to define a production that can describe any of the Roman numbers between five and eight, that is V VI VII VIII

```
N1 = "V" [ "I" ] [ "I" ] [ "I" ] .
N2 = "V" [ "I" [ "I" [ "I" ] ] ] .
N3 = "V" [ [ [ "I" ] "I" ] "I" ] .
N4 = "V" [ [ "I" "I" ] "I" ] .
```

Which of these productions is equivalent to N1? (There may be several.)

None	N2 *	N3 *	N4	Don't know

Which of these productions is ambiguous? (There may be several.)

None	N1 *	N2	N3	N4	Don't know

Which of these productions satisfies the LL(1) criteria? (There may be several.)

None	N1	N2 *	N3	N4	Don't know

Which of these productions cannot describe all four numbers? (There may be several.)

None	N1	N2	N3	N4 *	Don't know

Question A10 [9 marks]

Although a language like Parva may be described syntactically very accurately by a context-free grammar, there are aspects of the language that are distinctly "context-sensitive". These include restrictions like

* The labels used in a single switch statement must all be unique and distinct.

* Each identifier in a program is associated with a particular compile-time scope.

* The number of formal parameters specified for a function must match the number of actual arguments supplied when calling the function.

Suggest three other context-sensitive features that you have encountered. Which of these six features **cannot** be handled with reference to the familiar symbol table that is indexed by the identifiers introduced in declarations?

* Cannot assign boolean value to an integer variable (etc)

* Cannot mix booleans and integers willy-nilly

* must declare variable before using them (and other identifiable things, constants, functions

* Cannot redeclare an identifier in a nested block in which it is still in scope

* Can only have break statements within loops (or in switch clauses)

* etc

Cannot check on break statements using the symbol table (or would be perverse to do so), or the labels in switch.

Question A11 [4 marks]

Consider the following simple Parva program (which is not supposed to do anything useful, if it does anything at all!)

```
1      void main() { ;
2        const last = 122;;;
3        char c = 'a';
4        while (c < last) {
5          final int j = i + 1;
6          writeLine(i + j);
7          c++;
8        };
9      } // main
```

Which of the following claims is true and which is false?

This is a perfectly valid program	True *	False	Don't know
This program is invalid because you cannot add the value of a variable of type final to the value of a variable not of type final (line 6).	True	False *	Don't know
Marking a variable final means that its value cannot be altered. Each time you go around the loop you are trying to alter j, and this makes the program invalid (line 5).	True	False *	Don't know
The program is invalid because you cannot compare the character c with the constant last (line 4).	True	False *	Don't know
The program is invalid because semicolons are incorrectly placed.	True	False *	Don't know

Question A12 [4 marks]

What conclusion would you reach when you analyse the following Parva code:

```
do
  while ( Condition1 ) ;
while ( Condition2 ) ;
```

A do-while loop followed by a while loop	True	False *	Don't know
A while loop nested within a do-while loop	True *	False	Don't know
Either : the code seems to be ambiguous	True	False *	Don't know
Completely unacceptable, incorrect code	True	False *	Don't know

Question A13 [3 marks]

Part of the grammar for the *Primary* production for Parva - stripped of attributes to save space - reads as follows:

```
| "Clone"   "(" Expression ")"
| "Members" "(" Expression ")"
| "Equals"  "(" Expression "," Expression ")"
| "IsEmpty" "(" Expression ")"
| "Length"  "(" Expression ")"
| "Copy"    "(" Expression ")"
| "Equal"   "(" Expression "," Expression ")"
```

Might it have been better to write this part of the grammar with *Designator* everywhere that you currently see *Expression*, given that these operations all apply either to sets or to arrays? Justify the use of *Expression* by giving an example where the use of *Designator* could not have achieved a desired effect.

* You might want, for example `IsEmpty(a + b)` or `Clone(a * b)` or `Members(Clone({1,2}))`

* Expressions can have sets as operands, with + - * and / operators in particular

Question A14 [12 marks]

The following code is to be found in the Parva compiler

```
Type<out int type>
=  BasicType<out type>
   [ "[]"                          (. if (type != Types.noType) type++; .)
   ] .

BasicType<out int type>            (. type = Types.noType; .)
=    "int"                         (. type = Types.intType; .)
   | "bool"                        (. type = Types.boolType; .)
   | "char"                        (. type = Types.charType; .)
   | "set"                         (. type = Types.setType; .)
   | "charset"                     (. type = Types.charsetType; .) .


VarList<StackFrame frame, int type, bool cannotAlter>
=  OneVar<frame, type, cannotAlter>
   { WEAK "," OneVar<frame, type, cannotAlter> } .

OneVar<StackFrame frame, int type, bool cannotAlter>
                                   (. int expType;
                                      Entry var = new Entry(); .)
=  Ident<out var.name>             (. var.kind = Kinds.Var;
                                      var.type = type;
                                      var.cannotAlter = cannotAlter;
                                      var.offset = frame.size;
                                      frame.size++; .)
   (   AssignOp                    (. CodeGen.LoadAddress(var); .)
       Expression<out expType>     (. if (!Assignable(var.type, expType))
                                        SemError("incompatible types in assignment");
                                      CodeGen.Assign(var.type); .)
     |                             (. if (cannotAlter)
                                        SemError("defining expression required"); .)
   )                               (. Table.Insert(var); .) .
```

For each of the nonterminals that appear above, indicate whether the compiler is making use of synthesized or inherited attributes (or both, or neither).

Type	Synthesized *	Inherited	Both	Neither	Don't know
BasicType	Synthesized *	Inherited	Both	Neither	Don't know
VarList	Synthesized	Inherited *	Both	Neither	Don't know
OneVar	Synthesized	Inherited *	Both	Neither	Don't know
Ident	Synthesized *	Inherited	Both	Neither	Don't know
Expression	Synthesized *	Inherited	Both	Neither	Don't know

Question A15 [9 marks]

Consider the following attempt at writing a Parva program to illustrate identifier scope

```
void main ()                // point a
{ //  Demonstrate scope
   int i, j;                // point b
   {  int a, b;
      {  int x, y;
      }
      {  int x, z = x;      // point c
      }
   }
   int x, y;                // point d

} // main
```

a Table would be as in diagram	E
b Just after j has been declared table would be as in diagram:	F
c Just after z has been defined table would be as in diagram:	H
d Just after y has been declared table would be as in diagram:	J

Identify which of the diagrams below best represents the state of a symbol table at each of the points marked **a, b c** and **d** in this source code. The intersections marked O represent "scope nodes" as used in the Parva compiler in your exam kit. The first one, at point **a**, has been done for you: the diagram is **E**.

```
A   top -----> main ---> i -----> j -----> sentinel

B   top-o-----> x -----> z -----------+
        |                             |
        o-----> x -----> y -----------+
        |                             |
        o-----> a -----> b -----------+
        |                             |
        o-----> i -----> j -----------+
        |                             |
        o-----> main -----------------+---> sentinel

C   top-o-----> x -----> y ---> z ----+
        |                             |
        o-----> a -----> b -----------+
        |                             |
        o-----> i -----> j -----------+
        |                             |
        o-----> main -----------------+---> sentinel

D   top-o-----> x -----> y -----------+
        |                             |
        o-----> i -----> j -----------+
        |                             |
        o-----> main -----------------+---> sentinel

E   top-o-----> main -------------------> sentinel
```

```
F   top--o-----> i -----> j -------------+
         |                               |
         o-----> main -------------------+---> sentinel

G   top--o-> main ---> i ---> j ---> x ---> y ---> sentinel

H   top--o-----> x -----> z -------------+
         |                               |
         o-----> a -----> b -------------+
         |                               |
         o-----> i -----> j -------------+
         |                               |
         o-----> main -------------------+---> sentinel

I   top--o-----> x -----> y ------> z ---+
         |                               |
         o-----> x -----> y -------------+
         |                               |
         o-----> a -----> b -------------+
         |                               |
         o-----> i -----> j -------------+
         |                               |
         o-----> main -------------------+---> sentinel

J   top--o-----> i -----> j ---> x ---> y --+
         |                                  |
         o-----> main ----------------------+---> sentinel
```

Question A16 [5 marks]

Complete the code (with addresses and variable offsets) that would be generated for the program in Question A15, given again on the right.

```
0   FHDR
1   CALL  4
3   HALT
4   DSP  10     // space for variables
6   LDA  11     //
    LDA  10     //
    LDV         //
    STO         //


    RETV
```

```
void main ()
{ //  Demonstrate scope
   int i, j;
   {  int a, b;
      {  int x, y;
      }
      {  int x, z = x;
      }
   }
   int x, y;
} // main
```

Question A17 [3 marks]

Close perusal of the code for *OneVar* in question 14 might suggest that inserting a variable's entry into the symbol table could take place earlier, as follows

```
OneVar<StackFrame frame, int type, bool cannotAlter>
                                      (. int expType;
                                         Entry var = new Entry(); .)
=  Ident<out var.name>                (. var.kind = Kinds.Var;
                                         var.type = type;
                                         var.cannotAlter = cannotAlter;
                                         var.offset = frame.size;
                                         frame.size++;
                                         Table.Insert(var); .)    // ******************** here, earlier ?
   (   AssignOp                       (. CodeGen.LoadAddress(var); .)
       Expression<out expType>        (. if (!Assignable(var.type, expType))
                                           SemError("incompatible types in assignment");
                                         CodeGen.Assign(var.type); .)
     |                                (. if (cannotAlter)
                                           SemError("defining expression required"); .)
   ) .                                                            // *********** rather than here, later
```

Why would this earlier insertion, in fact, not be a good idea?

Consider, for example `int i = i + 4;`
We are declaring i, and also trying to define it using the expression `i + 4`, which is undefined.
So enter it into the symbol table after the statement is complete.

Question A18 [4 marks]

Short-circuit evaluation of expressions is prescribed for many computer languages. Give the two logical identities that lie at the heart of this idea.

`A AND B ≡ if A then B else FALSE`	`if NOT A then FALSE else B`
`A OR B ≡ if A then TRUE else B`	

Question A19 [5 marks]

Combined with suitable PVM opcodes, short-circuit evaluation of AND is achieved in Parva by code generated by the following familiar production (the production to handle short-circuit OR would be similar).

```
AndExp<out int type>               (. int type2;

                                      Label shortcircuit = new Label(!known); .)
=  EqlExp<out type>
   { "&&"                          (. CodeGen.BooleanOp(shortcircuit, CodeGen.and); .)
     EqlExp<out type2>             (. if (!IsBool(type) || !IsBool(type2))
                                         SemError("Boolean operands needed");
                                      type = Types.boolType; .)
   }                               (. shortcircuit.Here(); .)
.
```

Give the code for the production that would replace this one if you wanted to generate PVM code that did **not** use short-circuit evaluation, but instead was required to evaluate every *EqlExp* in the *Expression*.

```
AndExp<out int type>                (. int type2; .)


=  EqlExp<out type>

   { "&&" EqlExp<out type2>         (. if (!IsBool(type) || !IsBool(type2))

                                          SemError("Boolean operands needed");

                                       type = Types.boolType;

                                       CodeGen.BinaryOp(CodeGen.and); .)

   } .
```

Question A20 [4 marks]

After much argument it has been decided to extend Parva still further, adding a *loop-end* statement defined by

```
LoopStatement = "loop" { Statement } "end" ";" .
```

the semantics of which are simply to repeat the statement sequence between *loop* and *end* indefinitely. To be of any use, clearly, one or more of these statements might incorporate a *BreakStatement*, but you do not have to insist on that. A CSC 301 student has tackled this addition and has got as far as suggesting the production

```
LoopStatement<StackFrame frame>              (. Label loopExit = new label(!known);

                                                Label loopContinue = new Label(known);

                                             .)

=  "loop"

   {

   Statement<frame, loopExit, loopContinue>

   }                                         (. CodeGen.Branch(loopContinue);

                                                loopExit.Here(); .)

   "end"

   WEAK ";" .
```

Unfortunately this does not quite work, even though Coco might be happy to compile it. What must be added to make it work? (Make the changes on the code above.)

Question A21 [6 marks]

As a test case, the CSC 301 class has been asked to investigate the code that should be generated by the following noisy program

```
1    void main() {
2      write("Parrot says");
3      loop
4        write("Pretty");
5        continue;
6        write("Polly");
7        break;
8        continue;
9        break;              // partially generated
10     end;                  // fully generated
11   } // main
```

They are having trouble with the branch instructions. If the code generated by the program is as outlined here, what should it in fact look like just before the compiler reaches the *end* on line 10 (left column), and what should it look like when compilation is complete (right column)?

```
partially generated code

 0   FHDR
 1   CALL  4
 3   HALT
 4   DSP   0    // no locals
 6   PRNS  "Parrot says"
 8   PRNS  "Pretty"
10   BRN   8
12   PRNS  "Polly"
14   BRN   -1
16   BRN   8
18   BRN   14
```

```
fully generated code

 0   FHDR
 1   CALL  4
 3   HALT
 4   DSP   0    // no locals
 6   PRNS  "Parrot says"
 8   PRNS  "Pretty"
10   BRN   8
12   PRNS  "Polly"
14   BRN   22
16   BRN   8
18   BRN   22
20   BRN   8
22   RETV
```

Question A22 [10 marks]

Some language designers are never satisfied, or at least think they can do better. Members of the class might come up with several alternative suggestions for the syntax of this "infinite" loop. Classify each as an acceptable or unacceptable alternative. Hint - there may be more than one of each sort.

`LoopStatement = "loop" Statement .`	Acceptable *	Unacceptable	Don't know	
`LoopStatement = "loop" { Statement } ";" .`	Acceptable	Unacceptable *	Don't know	(LL(1))
`LoopStatement = "loop" Statement "end" .`	Acceptable *	Unacceptable	Don't know	
`LoopStatement = "loop" { Statement } "end" .`	Acceptable *	Unacceptable	Don't know	
`LoopStatement = "loop" { Statement } "end" ";" .`	Acceptable *	Unacceptable	Don't know	

Question A23 [4 marks]

In answers received to the November examination, several candidates suggested that the extension to allow a character set type should be achieved by using a grammar for *Primary* that included (call this ONE)

```
| [ "set"     ] "{" [ ElementList ] "}"
| [ "charset" ] "{" [ ElementList ] "}"
```

in place of the one in *Primary* suggested by several other candidates (call this TWO):

```
| [ "set" | "charset" ] "{" [ ElementList ] "}"
```

Comment on the following claims

ONE is necessary because two different set types will need two different productions.	True	False *	Don't know
Neither ONE nor TWO should be allowed to treat the key words set and charset as optional.	True	False *	Don't know
Either ONE or TWO, but not both, can be allowed to treat one of the key words set and charset as optional.	True *	False	Don't know
Both suggestions give rise to an LL(1) error, but this would be resolved as for the dangling else of the IfStatement and so would not matter.	True	False *	Don't know

Question A24 [4 marks]

The following code forms part of an attempt to interpret the PVM code generated for handling the evaluation of a set union, set difference, set intersection and symmetric difference. One of these *case* handlers is incorrect. Which one is it, and what should the corrected version be?

```
case PVM.union:          // union of two sets
  tempSet = GetSet(Pop());
  Push(AddSet(  tempSet.Union(GetSet(Pop()))  ));
  break;

case PVM.diff:           // difference of two sets
  tempSet = GetSet(Pop());
  Push(AddSet(  tempSet.Difference(GetSet(Pop()))  ));
  break;

case PVM.intsx:          // intersection of two sets
  tempSet = GetSet(Pop());
  Push(AddSet(  tempSet.Intersection(GetSet(Pop()))  ));
  break;

case PVM.xor:           // symmetric difference of two sets
  tempSet = GetSet(Pop());
  Push(AddSet(  GetSet(Pop()).SymDiff(tempSet)  ));
  break;
```

PVM.diff ________ is incorrect and the code should read as follows (operands in wrong order)

```
case PVM.             //

  tempSet = GetSet(Pop());

  Push(AddSet(  GetSet(Pop()).Difference(tempSet)  ));

  break;
```

Question A25 [5 marks]

In the Parva compiler as supplied in the exam kit and developed in November, sets may be compared for equality using code like

```
set a = set{0, 1, 3};
set b = set{3, 1, 0};
set c = set{2, 4, 6};
writeLine(Equals(a, b),  Equals(b, c),  ! Equals(b, a) );   // true, false, false
```

The designer of Parva might prefer that we apply the familiar `==` and `!=` operators to test sets for equality (meaning that they contain exactly the same elements). If this were allowed one would be able to write

```
writeLine(a == b,  b == c,  b != a);                        // true, false, false
```

Presumably this could be done by modifying the *EqlExp* production, which currently reads as follows:

```
EqlExp<out int type>                   (. int type2;
                                          int op;
                                          bool comparable = false; .)
=  RelExp<out type>
   [ EqualOp<out op> RelExp<out type2>  (. switch (op) {
                                            case CodeGen.ceq :
                                            case CodeGen.cne :
                                              comparable = Compatible(type, type2);
                                              break;
                                            case CodeGen.cin :
                                              comparable =    IsArith(type) && type2 == Types.setType
                                                          ||     type == Types.charType
                                                              && type2 == Types.charsetType;
                                              break;
                                          }
                                          if (!comparable)
                                            SemError("incomparable operand types");
                                          CodeGen.Comparison(op, type);
                                          type = Types.boolType; .)
   ]  .
```

Give the code for this change, in detail.

```
EqlExp<out int type>                   (. int type2;
                                          int op;
                                          bool comparable = false; .)
=  RelExp<out type>
   [ EqualOp<out op> RelExp<out type2>  (. switch (op) {
                                            case CodeGen.ceq :
                                            case CodeGen.cne :
                                              comparable = Compatible(type, type2);
                                              break;
                                            case CodeGen.cin :
                                              comparable =    IsArith(type) && type2 == Types.setType
                                                          ||     type == Types.charType
                                                              && type2 == Types.charsetType;
                                              break;
                                          }
                                          if (!comparable)
                                            SemError("incomparable operand types");
                                          if (IsSet(type) {                                      ////
                                            CodeGen.Equals();                                    ////
                                            if (op == CodeGen.cne) CodeGen.NegateBoolean();      ////
                                          else                                                   ////
                                            CodeGen.Comparison(op, type);                        ////
                                          type = Types.boolType; .)
   ] .
```

SECTION B - 2 questions. Answer Section B or Section C

Question B26 [6 marks]

In the Parva compiler as supplied in the exam kit and developed in November, sets may be initialized and manipulated using code like

```
set a = set{1, 2, 3};
charset vowels = charset{'a',  'e',  'i',  'o',  'u'};
a.Incl(10, 20, 30};
a.Excl(i, 2 * i, i * i);
```

The relevant parts of the compiler include the productions

```
ElementList<int elementType, bool inc>
=  Element<elementType>              (. CodeGen.IncludeOrExclude(inc); .)
   { "," Element<elementType>         (. CodeGen.IncludeOrExclude(inc); .)
   } .

Element<int elementType>             (. int type; .)
= Expression<out type>               (. if (!IsArith(type))
                                          SemError("arithmetic element needed");
                                        if (elementType == Types.charType && type != Types.charType)
                                          SemError("character element needed"); .) .
```

It would be useful to be able to specify multiple elements using a range notation, as illustrated by

```
charset letters  = charset{'a' .. 'z'};
charset digits   = charset{'0' .. '9'};
set nonLeapYears = {1999, 2001 .. 2003, 2005 .. 2007, 2009};
```

What should the following sets contain after construction? (The first one has been done for you.)

`set{ 1, 4 .. 7, 11 .. 12 }`	`{  1, 4, 5, 6, 7, 11, 12 }`
`set{ 0 .. 9 }`	`{ 0, 1,  2, 3, 4, 5, 6, 7, 8, 9 }`
`set{ 1, 7 .. 4 }`	`{ 1 }   or perhaps { 1, 4, 5, 6, 7 }`
`set{ 1 .. 4, 3 .. 5, 10 }`	`( 1, 2, 3, 4, 5, 10 }`
`charset{'A' .. 'C', 'Z' .. 'Z' }`	`{ 'A', 'B', 'C', 'Z' }`

Question B27 [20 marks]

Extend the Parva language, compiler and PVM to accomplish the extension for Question B26 (use the last pages of the question paper to do so).

This is straightforward once one sees the way to factor the production:

```
ElementList<int elementType, bool inc>
=  Range<elementType, inc>
   { "," Range<elementType, inc> }
.

Range<int elementType, bool inc>
=  Element<elementType>
   (   ".." Element<elementType>         (. CodeGen.IncludeOrExcludeRange(inc); .)
     |                                   (. CodeGen.IncludeOrExclude(inc); .)
   ) .
```

The new code generating function is very simple

```
public static void IncludeOrExcludeRange(bool inc) {
// Generates code to include/exclude all values between sos and tos inclusive
//  to/from a set with address just above that. (Process none if sos > tos)
  Emit(inc ? PVM.inclr : PVM.exclr);
} // CodeGen.IncludeOrExcludeRange
```

Finally, the new opcodes are straightforward. "Ranges" that are "wrong" such as 12 .. 3 have no effect, but we have chosen not to generate a runtime error. Other implementors might be more dictatorial.

```
case PVM.inclr:         // generate and include elements between sos and tos
  tos = Pop();
  sos = Pop();
  tempSet = GetSet(mem[cpu.sp]);
  for (loop = sos; loop <= tos; loop++) tempSet.Incl(loop);
  break;

case PVM.exclr:         // generate and exclude elements between sos and tos
  tos = Pop();
  sos = Pop();
  tempSet = GetSet(mem[cpu.sp]);
  for (loop = sos; loop <= tos; loop++) tempSet.Excl(loop);
  break;
```

SECTION C - 2 questions. Answer Section B or Section C

Question C28 [6 marks]

The Parva compiler as supplied in the exam kit, incorporating the ideas you should have developed in the 24 hour period before the examination, allows for functions to be "prototyped" before they are fully defined, opening up the possibility for a Parva compiler to handle mutually recursive functions:

```
void one();       // function prototype
void two(int x);  // function prototype

void main() {
  one();
  two(3);
} // main

void one() {
  two(1);
} // one

void two(int x) {
  if (x > 1) two(x-1);  // recursive call
} // two
```

Unfortunately this system will not detect several errors that users might make. Consider the following code

```
void one(int i);
void two(int a, bool b);
void three(char c, int n);
void four();
void five(int x);

void one(int i, int j) {}           // 1  wrong number of parameters

void two(int i, int j) {}           // 2  wrong names and mismatched types

void four(int j) {}                 // 3  wrong number of parameters

void four();                        // 4  can't use the prototype again here

void main() {
  one(3);
  one(4, 5);
  three('a', 4);
  five(3);                          // 5  five was never properly defined
}

void three(char ch, int nn) {}      // 6  wrong names

                                    // 7  four was never properly defined.
                                    //    since it was never called this may not matter
```

Identify at least three places in the code (with reasons) where errors slip past the compiler and remain undetected. Simply annotate the code above.

Question C29 [20 marks]

Function declarations in the system supplied to you are handled by the following code. Improve the Parva compiler so that it will detect errors like those you identified in Question C28 (use the last pages of the question paper to do so).

```
FuncDeclaration
                                        (. StackFrame frame = new StackFrame();
                                           Entry function = new Entry(); .)
=  "void" Ident<out function.name>      (. function.kind = Kinds.Fun;
                                           function.type = Types.voidType;
                                           function.nParams = 0;
                                           function.firstParam = null;
                                           Entry oldEntry = Table.Find(function.name);
                                           bool completing = !oldEntry.defined;
                                           if (!completing) Table.Insert(function);
                                           Table.OpenScope(); .)
   "(" FormalParameters<function> ")"   (. frame.size = CodeGen.headerSize + function.nParams; .)
   (                                    (. if (completing) {
                                             oldEntry.entryPoint.Here();
                                             oldEntry.defined = true;
                                           }
                                           else {
                                             function.entryPoint = new Label(known);
                                             function.defined    = true;
                                           }
                                           if (function.name.ToUpper().Equals("MAIN")
                                               && !mainEntryPoint.IsDefined()
                                               && function.nParams == 0) {
                                             mainEntryPoint.Here();
                                           } .)
       Body<frame>
     | ";"                              (. function.entryPoint = new Label(!known);
                                           function.defined = false; .)
   )                                    (. Table.CloseScope(); .) .
```

This is a fairly tricky thing to get exactly right. The question is intended as a discriminator.

The places where the code needs amending are highlighted below/ Note that we need to add a method to the table handler (called at the end of compilation) to check that all functions have been completely defined. If we also keep track of which functions are actually called we can perhaps issue warning against "unused" prototypes.

```
Parva
                                        (. CodeGen.FrameHeader();        // no arguments
                                           CodeGen.Call(mainEntryPoint); // forward, incomplete
                                           CodeGen.LeaveProgram(); .)    // return to O/S
= {   FuncDeclaration } EOF             (. if (!mainEntryPoint.IsDefined())
                                             SemError("missing Main function");
                                           CodeGen.FixStrings();
                                           Table.CheckFunctionsDefined(); .) .           /////

FuncDeclaration                         (. StackFrame frame = new StackFrame();
                                           Entry function = new Entry(); .)
=  "void" Ident<out function.name>      (. function.kind = Kinds.Fun;
                                           function.type = Types.voidType;
                                           function.nParams = 0;
                                           function.firstParam = null;
                                           Entry oldEntry = Table.Find(function.name);
                                           bool completing = !oldEntry.defined;
                                           if (!completing) Table.Insert(function);
                                           Table.OpenScope(); .)
   "(" FormalParameters<function> ")"   (. frame.size = CodeGen.headerSize + function.nParams; .)
   (                                    (. if (completing) { // patch prototype entry
                                             oldEntry.entryPoint.Here();
                                             oldEntry.defined = true;
                                             if (Mismatch(oldEntry, function))                 /////
                                               SemError("parameter list differs from prototype"); /////
                                           }
                                           else { // standard declaration
                                             function.entryPoint = new Label(known);
                                             function.defined    = true;
                                           }
```

```
                                              if (function.name.ToUpper().Equals("MAIN")
                                                  && !mainEntryPoint.IsDefined()
                                                  && function.nParams == 0) {
                                                mainEntryPoint.Here();
                                              } .)
        Body<frame>
      | ";"                                 (. if (oldEntry.kind == Kinds.Fun && oldEntry.defined)      /////
                                                SemError("already defined");                             /////
                                              if (completing)                                            /////
                                                SemError("function body expected");                      /////
                                              else {
                                                function.entryPoint = new Label(!known);
                                                function.defined = false;
                                              }  .)
    )                                       (. Table.CloseScope(); .) .
```

Note the way in which the original (prototype) formal parameter list is checked against the defining parameter list. This might be too severe - the number and types of the parameters must match exactly, but the names could be left unchecked. Or could they - I leave this to the reader to ponder and experiment!

```
static bool Mismatch(Entry proto, Entry actual) {
// Returns true if the proto and actual parameter lists do not match
// used to check function prototype completion
  if (proto.nParams != actual.nParams) return true;
  Entry pp = proto.firstParam;
  Entry ap = actual.firstParam;
  for (int i = 1; i <= proto.nParams; i++) {
    if (pp.type != ap.type || !Equals(pp.name, ap.name)) return true;
    pp = pp.nextInScope; ap = ap.nextInScope;
  }
  return false;
} // Mismatch
```

The *Entry* class needs enhancement:

```
class Entry {
// All fields initialized, but are modified after construction (by semantic analyser)
  public int     kind        = Kinds.Var;
  public string  name        = "";
  public int     type        = Types.noType;
  public int     value       = 0;       // constants
  public int     offset      = 0;       // variables
  public bool    declared    = true;    // true for all except sentinel entry
  public Entry   nextInScope = null;    // link to next entry in current scope
  public int     nParams     = 0;       // functions
  public Label   entryPoint  = new Label(false);
  public Entry   firstParam  = null;
  public bool    cannotAlter = false;   // false except when marked immutable
  public bool    defined     = true;    // true unless it marks a function prototype           /////
  public bool    called      = false;   // true if this function has been called               /////
} // end Entry
```

and the function for checking whether all functions have been properly defined might look like this:

```
  public static void CheckFunctionsDefined() {
  // At end of compilation, check that all prototypes were completed
    Scope scope = topScope;
    Entry entry = scope.firstInScope;
    while (entry != sentinelEntry) {
      if (entry.kind == Kinds.Fun && !entry.entryPoint.IsDefined())
        if (entry.called)
          Parser.SemError("Function " + entry.name + " was called, but never defined");
        else
          Parser.Warning("Function " + entry.name + " was never fully defined or called");
      entry = entry.nextInScope;
    }
  } // Table.CheckFunctionsDefined
```