RHODES UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE

EXAMINATIONS: NOVEMBER 2016

Computer Science 301

PAPER 1 - Translators - Solutions

Internal Examiner: Prof P.D. Terry

MARKS: 180
DURATION: 4 hours

External Examiner: Prof M. Kuttel

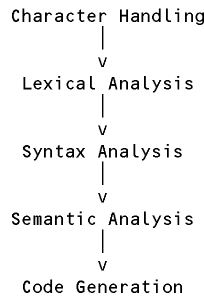
Answer all questions. Answers may be written in any medium except red ink.

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

Section A: Conventional questions [95 marks]

QUESTION A1 **[6 + 4 + 4 = 14 marks]**

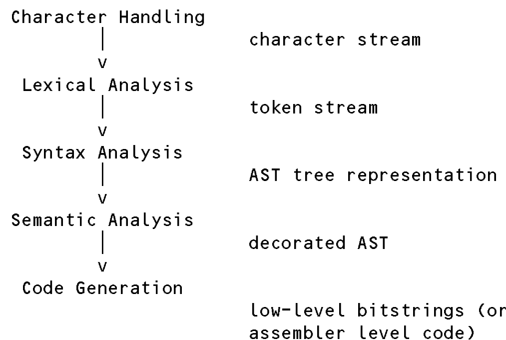
A1 A compiler is often described as incorporating several "phases", as shown in the diagram below:



These phases are often developed in conjunction with an Error Handler and a Symbol Table Handler.

To place this all in context:

At the Character Handler level the program is essentially a sequence of characters. After Lexical Analysis it is effectively a stream of tokens. Syntax Analysis conceptually (or even in practice) builds a tree, and Semantic Analysis decorates the tree to give a form of intermediate code. Code Generation produces a file of low-level bitstrings, or possibly low-level assembler like code.



(a) Suggest - perhaps by example - the sort of errors that might be detected in each of the phases of the compilation process. [6 marks]

There is plenty of scope for variation in the answer to this question! At the character handling level the errors would arise from incomplete, missing, or corrupt files. At the lexical analysis stage one might fail to recognize a valid token at all (for example, if one came across an isolated \ in a Parva program, or had a comment that "opened" but did not "close"). At the syntax analysis stage errors such as improperly formed statements or missing punctuation might arise. Semantic analysis might throw up errors such as the use of undeclared identifiers or mismatched types in the operands of expressions. Code generation might lead to further file handling errors. For the sorts of "interpretive" compilers the class had seen, other code generation errors could arise from trying to generate too much code for the simulated memory of the virtual machine to handle.

(b) The Parva compiler studied in the course is an example of what is often called a "one pass incremental compiler" - one in which the various phases summarized above are interleaved, so that the compiler makes a single pass over the source text of a program and generates code as it does so, without building an explicit AST. Discuss briefly whether the same technique could be used to compile one *class* of a C# program (consider the features of C# and Parva that support the use of such a technique, and identify features that seem to act against it). [4 marks]

The insight called for here is that in C# one does not have to "declare before use" where methods are concerned, which is very difficult to handle on a single pass without building an AST (which effectively allows for an easy second pass).

(c) The Parva compiler supports the integer, character and Boolean types. Integer and character constants (literals) are represented in the usual way, for example 1234 (decimal) and 'X', respectively.

Suppose we wished to allow hexadecimal and binary representations of integer literals as well, for example 012FH and 01101% respectively. Which of the phases of compilation identified here would this affect, which would it not affect, and why? [4 marks]

Essentially it affects only lexical analysis (for recognizing the alternative character strings permissible) and the small amount of semantic analysis needed to convert such strings to their corresponding integer values. In solutions submitted by candidates, a great many may confuse "types" and "values". Integer values can be represented in various ways - for example 123, 07FH, 0111% - but all these are values of the same type, so the grammar would not require the introduction of special compatiblity constraints, or the generation of special opcodes, for example. In a Cocol grammar for Parva the necessary changes would amount to the following - however, all this detail was not required.

```
CHARACTERS
  digit       = "0123456789" .
  hexDigit    = digit + "ABCDEF"
  binDigit    = "01"
  ...
TOKENS
  decNumber   = digit { digit } .
  hexNumber   = digit { hexdigit } "H" .
  binNumber   = bindigit { bindigit } "%" .
  ...

PRODUCTIONS
  ...

  IntConst<out int value>            (. int base = 10; .)
  = (   decNumber                    (. base = 10; .)
      | hexNumber                    (. base = 16; .)
      | binNumber                    (. base = 2; .)
    )                                (. try {
                                          value = Convert.ToInt32(token.val, base);
                                        } catch (Exception) {
                                          value = 0; SemError("number out of range");
                                        } .).
```

QUESTION A2 **[6 + 6 = 12 marks]**

A2 Please combine the solutions to (a) and (b) in joint CHARACTERS and TOKENS definitions in Cocol.

(a) Write down the definition of a token that will match all the possible mother-tongue words that have each vowel appearing exactly once, and in the correct order (vowels are a, e, i, o and u; an example of such a word in English is "facetious"). [6 marks]

(b) Currency values are sometimes expressed in a format exemplified by

R12,345,101.99

where, for large amounts, the digits are grouped in threes to the left of the point. Exactly two digits appear to the right of the point. The leftmost group might only have one or two digits. Add the definition of a token that would match a valid currency value. [6 marks]

Your solution should not match an incorrect representation that has leading zeroes, like R001,123.00

```
CHARACTERS
   nonzero = "123456789" .
   digit   = nonzero + "0" .
   vowel   = "aeiouAEIOU".
   letter  = "ABCDEFGHIJKLMNOPQRSTUVWXYZ" +
             "abcdefghijklmnopqrstuvwxyz" .
   consonant = letter - vowel .
TOKENS
   randAmount    = "R" (   "0"
                         | nonzero [ digit [ digit ]] { "," digit digit digit }
                       )
                   "." digit digit .
   allVowelsOnce = { consonant } ( "A" | "a" )
                   { consonant } ( "E" | "e" )
                   { consonant } ( "I" | "i" )
                   { consonant } ( "O" | "o" )
                   { consonant } ( "U" | "u" )
                   { consonant } .
```

QUESTION A3 **[10 marks]**

Here is a Cocol description of simple mathematical expressions:

```
COMPILER Expression  $NC  /* expr.atg */
CHARACTERS
  digit      = "0123456789" .
TOKENS
  Number     = digit { digit } .
PRODUCTIONS
  Expression = Term { "+" Term  | "-" Term } .
  Term       = Factor { "*" Factor | "/" Factor } .
  Factor     = Number | "(" Expression ")" .
END Expression.
```

Two CSC 301 students were arguing very late at night / early in the morning about a prac question which read "extend this grammar to allow you to have leading unary minus signs in expressions, as exemplified by `-5 * (-4 + 6)`, and make sure you get the precedence of all the operators correct". One student suggested that the productions should be changed to read (call this GRAMMAR A)

```
Expression = [ "-" ] Term { "+" Term  | "-" Term } .     /* change here */
Term       = Factor { "*" Factor | "/" Factor } .
Factor     = Number | "(" Expression ")" .
```

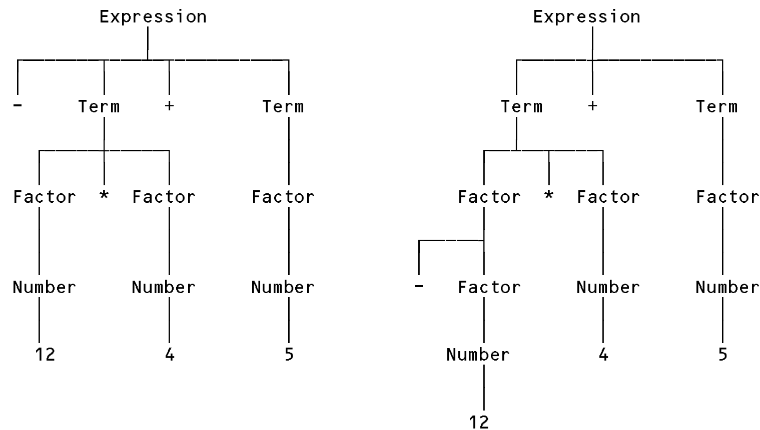
while the other suggested that the correct answer would be (call this GRAMMAR B)

```
Expression = Term { "+" Term  | "-" Term } .
Term       = Factor { "*" Factor | "/" Factor } .
Factor     = Number | "-" Factor | "(" Expression ")" .  /* change here */
```

By drawing the parse trees for the string `- 12 * 4 + 5` try to decide which student was correct. Or were they both correct? If you conclude that the grammars can both accept that particular example expression, are the grammars really equivalent, or can you give an example of an expression that GRAMMAR A would accept but GRAMMAR B would reject (or vice-versa)? [10 marks]

*Both grammars are correct, in the sense that they can derive these sorts of strings with the correct precedence attached to the operators. The leading unary minus in the first grammar will be bound to the first term only, and because (-a)*b = -(a*b) = a*(-b) the arithmetic meaning would be retained. In the second grammar the meaning of -a*b would be explicitly (-a)*b, which is the same. (Quite unbelievably, several students may claim that this would not be the case. Where, one wonders, did they learn arithmetic!) The subtle difference is that the second grammar would allow expressions like a * - b or a + - b or even a - - - - b, which the first will not do. An expression like a - - b would be taken to mean a - (-b) = a + b, so all would be well. It is of interest to record that C# incorporates the ideas of the second grammar, while Pascal/Modula incorporate the first ideas.*

The parse trees are as follows



QUESTION A4 **[8 + 2 + 16 + 2 + 4 = 32 marks]**

The contents page of a very useful textbook (if you can find a copy) begins as follows: (contents.txt)

```
All you need to know to be able to pass your compiler examination.

                              by

                          Pat Terry.

Chapter 1  Bribery is unlikely to succeed.

Chapter 2  Understand the phases of compilation.
    2.1    Lexical and syntactic analysis are easily confused
    2.2    Constraint analysis involves the concept of type
    2.3    Code generation for the PVM is a breeze

Chapter 3  Get clued up on grammars.
    3.1    Terminals
    3.2    Sentences and sentential forms
    3.3    Productions
    3.4    EBNF and Cocol
    3.5    Ambiguity is bad news
```

The following Cocol grammar attempts to describe this contents page (and others like it - there may be further chapters and further subsections, of course, and some components are optional):

```
COMPILER Contents                            // contents.atg
/* Describe the contents pages of a book
   P.D. Terry, Rhodes University, 2016 */

CHARACTERS
  uLetter = "ABCDEFGHIJKLMNOPQRSTUVWXYZ" .
  lLetter = "abcdefghijklmnopqrstuvwxyz" .
  letter  = uLetter + lLetter .
  digit   = "0123456780" .

TOKENS
  word    = letter { letter } .
  number  = digit { digit } .
  section = digit { digit } "." digit { digit } .

IGNORE CHR(0) .. CHR(31)

PRODUCTIONS // Version 1
  Contents   = [ Title [ "by" Author [ Date ] ] ] { Chapter } .
  Title      = word { word } "." .
  Author     = word { word } "." .
  Date       = "(" number ")" .
  Chapter    = "Chapter" number Title { Subsection } .
  Subsection = section word { word } .
END Contents.
```

(a) Is this an LL(1) grammar? Justify your answer, don't just guess! [8 marks]

Yes, it is an LL(1) grammar. To justify this we could either transform the grammar to eliminate the meta-brackets (tedious) or, more directly, look at the nullable components within the productions, as the grammar does not explicitly make use of alternation symbols.

There are nullable components of the form { `word` } *in the productions for* `Title`, `Author` *and* `Subsection`. *In the first two of these the FIRST set for the nullable component contains* `word`, *while the FOLLOW set contains* ".", *so there is no problem with those. In the case of* `Subsection` *the FIRST set of the nullable component contains only* `word`, *while the FOLLOW set, contains* `EOF`, `section` *and* `"Chapter"`, *so there is again no problem.*

The productions for `Chapter` *and* `Contents` *contain various nullable components. In the case of* `Chapter` *there is no problem, as the relevant FIRST and FOLLOW sets are* { `section` } *and* { `EOF`, `"Chapter"` }. `Contents` *is slightly more complicated, but by inspection one can again see that there will be no problems, for similar reasons.*

(b) The above grammar allows the *Contents* to be completely empty. Comment on the following attempt to change it so that (i) if the *Title* appears the list of chapters is optional but (ii) if the *Title* is absent there must be at least one *Chapter*. [2 marks]

```
PRODUCTIONS // Version 1 modified
  Contents   =   [ Title [ "by" Author [ Date ] ] ] Chapter { Chapter }
               | Title [ "by" Author [ Date ] ] { Chapter } .
```

This is a valid way of expressing the constraint, but it clearly renders the grammar non-LL(1) as both options for `Contents` *have* `word` *in their FIRST sets, thus contravening "Rule 1". A much better change would be*

```
PRODUCTIONS // Version 1 modified rather better
  Contents   =   Title [ "by" Author [ Date ] ] { Chapter } | Chapter { Chapter }
```

(c) How would you add actions to the *original grammar* (Version 1) above so as to develop a system that could tell you the title of the chapter with the greatest number of subsections and also issue a warning if the contents turns out to be completely empty? [16 marks]

A simple solution using a few static variables will suffice, similar to many examples seen in the practical course. A C# version follows:

```
using Library;

COMPILER Contents $NCF
/* Describe the contents pages of a book
   P.D. Terry, Rhodes University, 2016 */

static string chapter, maxChapter = "";
static int sections, maxSections = 0;

CHARACTERS
  uLetter = "ABCDEFGHIJKLMNOPQRSTUVWXYZ" .
  lLetter = "abcdefghijklmnopqrstuvwxyz" .
  letter  = uLetter + lLetter .
  digit   = "0123456780" .

TOKENS
  word    = letter { letter } .
  number  = digit { digit } .
  section = digit { digit } "." digit { digit } .

IGNORE CHR(0) .. CHR(31)

PRODUCTIONS // Version 1
  Contents               (. bool hasTitle = false; .)
  = [ Title              (. hasTitle = true; .)
      [ "by" Author [ Date] ]
    ]
    { Chapter            (. if (sections > maxSections) {
                                maxSections = sections;
                                maxChapter = chapter;
                              } .)
    }                    (. IO.WriteLine("Chapter with most subsections: " + maxChapter);
                             if (!hasTitle && maxSections == 0)
                               Warning("Completely empty contents"); .) .
```

```
    Title
    = word                (. chapter = token.val; .)
      { word              (. chapter = chapter + " " + token.val; .)
      } "." .

    Author
    = word { word } "." .

    Date
    = "(" number ")" .

    Chapter               (. sections = 0; .)
    = "Chapter" number
       Title { Subsection   (. sections++; .)
       } .

    Subsection
    = section word { word } .

  END Contents.
```

(d) Here are two other possibilities for the set of productions for this system:

```
PRODUCTIONS // Version 2
  Contents   = [ Sentence [ "by" Sentence [ "(" number ")" ] ] ] { Chapter } .
  Sentence   = Words "." .
  Words      = word { word } .
  Chapter    = "Chapter" number Sentence { Subsection } .
  Subsection = section Words .
END Contents.

PRODUCTIONS // Version 3
  Contents   = [ Words "." [ "by" Words "." Date ] ] { Chapter } EOF .
  Chapter    = "Chapter" number Words "." { section Words } .
  Words      = word { word } .
  Date       = [ "(" number ")" ] .
END Contents.
```

If I were to claim that these grammars are "equivalent" I would be using the word "equivalent" in a special way. What is meant by the statement "two grammars are equivalent"? [2 marks]

Two grammars G1 and G2 are equivalent if L(G1) is identical to L(G2), that is to say, if the set of strings defined by the one grammar is identical to the set of strings defined by the second one. This does not demand that the sentences themselves have identical parse trees, however.

(e) Even though the sets of productions are equivalent, a developer might have reasons for preferring one set over the others. Which of these sets do you consider to be the "best", and why? [4 marks]

The first set has made far more use of non-terminals named to suggest the semantic ideas that could be useful in identifying the various components found on a contents page. It does this at the expense of "repeating" the definition of Title to give the one for Author. However, if one were to develop a system further (as is done in (c)) it becomes useful to have these semantic "hooks" available, even though syntactically they may appear redundant.

QUESTION A5 **[12 marks]**

Consider the description of the contents of a book in Question A4, using the productions in version 1.

Assume that you have `accept()` and `abort()` routines like those you used in this course, and a scanner `getSym()` that can recognise tokens that might be enumerated by

```
EOFSym, noSym, wordSym, numberSym, sectionSym, bySym, periodSym, chapterSym, lparenSym, rparenSym
```

How would you complete the *parser* routines below? There is no need to incorporate the actions required in (c) of Question A4 - simply show the syntax analysis. A spaced copy of this system appears in the machine readable version of the paper, which you are invited to complete and submit. [12 marks]

You are *not* required to give any of the code for the *scanner*.

These sorts of parser are very easily written. There are, sadly, various traps that people fall into - such as inserting calls to `getSym()` *where they are not needed, or leaving them out in places where they are needed!*

```
static void Contents () {
// Contents = [ Title [ "by" Author [ Date ] ] ] { Chapter } .
  if (sym.kind = wordSym) {
    Title();
    if (sym.kind == bySym) {
      GetSym();
      Author();
      if (sym.kind == lparenSym) Date();
    }
  }
  while (sym.kind == chapterSym) Chapter();
}

static void Title () {
// Title = word { word } "." .
  Accept(wordSym, "word expected");
  while (sym.kind == wordSym) GetSym();
  Accept(periodSym, " . expected");
}

static void Author () {
// Author = word { word } "." .
  Accept(wordSym, "name expected");
  while (sym.kind == wordSym) GetSym();
  Accept(periodSym, " . expected");
}

static void Date () {
// Date = "(" number ")" .
  Accept(lparenSym, "( expected");
  Accept(numberSym, "year expected");
  Accept(rparenSym, ") expected");
}

static void Chapter () {
// Chapter = "Chapter" number Title { Subsection } .
  Accept(chapterSym, "Chapter expected");
  Accept(numberSym, "number expected");
  Title();
  while (sym.kind == sectionSym) Subsection();
}

static void Subsection () {
// Subsection = section word { word } .
  Accept(sectionSym, "section number expected");
  Accept(wordSym, "word expected");
  while (sym.kind == wordSym) GetSym();
}
```

QUESTION A6 **[9 + 2 + 4 = 15 marks]**

A6 Brinch Hansen, an influential Danish computer scientist in the 1980s, incorporated an extended form of the *WhileStatement* in his experimental language Edison, instead of the usual one. Essentially this was defined as follows:

```
WhileStatement  =  "while" "(" Condition ")" Statement
                   { "or" "(" Condition ")" Statement } .
```

The dynamic semantics of this unfamiliar construct are that *Conditions* are evaluated one at a time in the order written until one is found to be true, when the corresponding *Statement* is executed, after which the process of evaluating conditions is repeated from the top. If no *Condition* is true, the loop terminates.

(a) Assuming that you want to incorporate this into Parva, and to target the PVM, how would you attribute the production to handle code generation? [9 marks]

```
WhileStatement<StackFrame frame>              (. Label loopContinue = new Label(known);
                                                 Label falseLabel   = new Label(!known);
                                                 Label loopExit     = new Label(!known; .)
= "while" "(" Condition ")"                   (. CodeGen.BranchFalse(falseLabel); .)
   Statement<frame, loopExit, loopContinue>   (. CodeGen.Branch(loopContinue); .)
   {                                          (. falseLabel.Here();
                                                 falseLabel = new Label(!unknown); .)
      "or" "(" Condition ")"                  (. CodeGen.BranchFalse(falseLabel); .)
      Statement<frame, loopExit, loopContinue> (. CodeGen.Branch(loopContinue); .)
   }                                          (. falselabel.Here(); .)
                                              (. loopExit.Here() .) .
```

(b) In his original proposal, the keyword *or* was, in fact, *else*. Why can we not follow Brinch Hansen and use *else* in Parva as well? [2 marks]

For the simple reason that one would have a very nasty LL(1) problem. If the statement forming the body of a loop were a simple IfStatement without an "else" then the "else Statement" of the WhileStatement would be assumed to be part of this IfStatement, and not the next option in the Brinch Hansen WhileStatement.

(c) Here is a simple example of a Brinch Hansen *WhileStatement*. Write a "regular" Parva statement (or statement sequence) that would achieve the same effect. [4 marks]

```
int a, b, c;       // assume that these will have had values defined by the time
                   // the WhileStatement is reached
while (a > b) {
  write(a, b);
  --b;
}
or (b < 10) write(b);
```

This is a bit clumsy; two alternatives are offered below, Actually I doubt if I can think up a situation where I would want to use the Brinch Hansen loop structure. I'm not even sure he even used it himself!

```
int a, b, c;

while (a > b || b < 10) {
  if (a > b) {
    write(a, b);
    --b;
  }
  else write(b);
}
```

```
int a, b, c;
while (true)
  if (a > b) {
    write(a, b)
    --b;
  }
  elsif (b < 10)
    write(b);
  else break;
```

Section B: Parva with the addition of sets [85 marks]

QUESTION B7 **[6 marks]**

B7 Oh dear! This code below should never compile and run, and yet it does. What two checks has the compiler writer forgotten to make, and how should the system be corrected? [6 marks]

```
void main() {                         // egB7.pav
  set a, b = set{5, 9, 7 > 4};
  write(a);
} // main
```

There is (a) no check that the elements are arithmetic (Boolean elements are meaningless) and (b) set a is still undefined by the time the write statement is reached. The first check is easily added:

```
   Element                              (. int type; .)
   =  Expression<out type>              (. if (!IsArith(type))
*                                             SemError("arithmetic element needed"); .)
```

The second problem is more awkward. The heap for the sets has been primed with an empty set at index position 0 and so an undefined set variable will, by default, be initialised to this and probably remain undetected. But the compiler should either demand that a set be initialised explicitly when it is declared, or should trap an access to the fake entry on the heap. The first of these can be achieved by a modification to OneVar

```
    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 || IsSetType(var.type)
                                                 SemError("defining expression required"); .)
      )                                     (. Table.Insert(var); .) .
```

while the second suggestion would requre a simple change to the PVM

```
      public static IntSet GetSet(int i) {
      // Retrieves string i from the string pool
*       if (i <= 0 || i >= setHeap.Count) {
          ps = nullRef; return setHeap[0];
        }
        return setHeap[i];
      } // PVM.GetSet
```

QUESTION B8 **[36 marks]**

B8 The system as supplied to you allows for the operations of set definition, forming the union of two sets, including further elements into a set, testing a value for membership of a set, cloning (copying) a set, comparing two sets for equality, determining the number of elements a set contains, and writing a set in a simple way, as illustrated by the following otherwise rather pointless program:

```
void main() {                     // egB8A.pav
  set a, b, c, d;
  a = set{5, 9};
  b = set{7, 4, 9};
  b.Incl(9, 1);                   // set b is now {1, 4, 7, 9}
  c = a + b;                      // union (  {1, 4, 5, 7, 9} )
  d = Clone(b);                   // cloning a set to make a distinct copy
  bool e = 3 in b;                // test for membership (false)
  if (Equals(a, b))               // in this case they are not equal
    write("compiler error");
  write(Members(b));              // number of elements (4)
  write("set b is " , b);         //   set b is {1, 4, 7, 9}
} // main
```

Extend the compiler and interpreter to allow for the operations of forming the difference of two sets, excluding elements from a set, and testing for a set being empty, as illustrated in the following otherwise equally pointless program: [3 x 12 = 36 marks]

```
void main() {                     // egB8B.pav
  set a, b, c, d, e;
  a = set{5, 9};
  b = set{7, 4, 9, 1};
  c = b - a;                      // difference (  {1, 4, 7} )
  d = set{};                      // empty set
  write(IsEmpty(b), IsEmpty(d));  // false, true
  b.Excl(1, 7);                   //   set b is now {4, 9}
} // main
```

Any student who really came to terms with the problem in the 24 hour period should have little difficulty with these; it's just a matter of hacking the code.

The Excl statement is a simple variation on the Incl statement (both are variations of assignments), so that at the low level the PVM simply needs another opcode, very similar to the PVM.incl opcode

```
    case PVM.excl:           // exclude tos from a set
*     GetSet(mem[cpu.sp + 1]).Excl(Pop());
      break;
```

However, the ElementList method needs an important parameter addition, as the elements in such a list have either all to be added to a set, or excluded from it. ElementList is called from two places - one being in Primary where the elements will always be added:

```
          |                                      (. type = Types.setType; .)
           [ "set" ]                             (. int elementType = ElementType(type);
                                                    CodeGen.ConstructSet(); .)
*          "{" [ ElementList<elementType, true>
               ]
           "}"
```

The other place is in AssignOrCall

```
          |                                      (. if (!IsSetType(des.type))
                                                      SemError("set designator needed"); .)
*           "." ( "Incl" | "Excl"                (. inc = false; .)
            )                                    (. CodeGen.Dereference(); .)
*           "(" ElementList<inc>
            ")"                                  (. CodeGen.Pop(); .)
```

The revised ElementList production can make use of a generalised IncludeOrExclude code generator

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

To handle the set difference operation calls for a simple variation on the union operation.

```
   AddExp<out int type>                          (. int type2;
                                                    int op; .)
   =  MultExp<out type>
      { AddOp<out op>
        MultExp<out type2>                       (. if (IsArith(type) && IsArith(type2)) {
                                                      type = Types.intType;
                                                      CodeGen.BinaryOp(op);
                                                    }
                                                    else if IsSetType(type) && IsSetType(type2))
                                                      switch (op) {
                                                        case CodeGen.add : CodeGen.BinaryOp(CodeGen.uni); break;
*                                                       case CodeGen.sub : CodeGen.BinaryOp(CodeGen.dif); break;
                                                      }
                                                    else {
                                                      SemError("arithmetic operands needed");
                                                      type = Types.noType;
                                                    } .)
      }
   .
```

with the PVM.diff opcode defined by a simple variation on the PVM.uni operation:

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

To handle the query as to whether a set is empty is also fairly trivial. We add another option to the Primary production:

```
*     | "IsEmpty" "(" Expression<out type>       (. if (!IsSetType(type))
*                                                     SemError("not a set"); .)
*       ")"                                      (. CodeGen.IsEmpty();
*                                                   type = Types.boolType; .)
```

with the CodeGen method as

```
*    public static void IsEmpty() {
*    // Generates code to check for an empty set
*      Emit(PVM.empty);
*    } // CodeGen.IsEmpty
```

and the PVM.empty opcode defined by

```
*       case PVM.empty:         // is this set empty
*         Push(GetSet(Pop()).IsEmpty() ? 1 : 0);
*         break;
```

QUESTION B9 **[4 marks]**

B9 A tutor spotted a student writing Parva code that included a scan of a string of binary digits:

```
char ch;
read(ch);
while (ch in set{'0', '1'})
  read(ch);
```

"That will work", said the tutor, "but to save both space and time you would be better advised to write the code as follows"

```
set digits = {'0', '1'};
read(ch);
while (ch in digits)
  read(ch);
```

Why did the tutor say that the modification would save both space and time? [4 marks]

It should be pretty obvious that defining the set once, and not redefining it over and over again as the loop executes will save some time. What might not be so obvious is that redefining it on every pass is going to add another set to the heap, thus slowly consuming space as well.

QUESTION B10 **[5 marks]**

B10 The compiler supplied to you has the ability to mark variables "final" as was suggested in your last practical exercise. Surely, to all intents and purposes a declaration like

```
const x = 10;
```

has exactly the same effect as the declaration

```
final int x = 10;
```

because we should expect a compilation error if we were to follow either declaration with the statement:

```
x = x + 1;
```

"Not so" said the language lawyer. "While it is true that in this case both would generate the same sort of error, there are subtle differences in the way these declarations can be used".

Lawyers produce evidence. Identify and explain the alleged differences by considering the PVM code that would be generated for each of the following snippets [5 marks]

```
(a)  int y;                    (b)  int y;
     const x = 10;                  final int x = 10;
     y = x + 15;                    y = x + 15;
```

A "const" definition inserts an entry into the symbol table that allows the corresponding constant to be "inlined" in later code. Thus, for example

```
const x = 10;
int   y = x + 15;
```

would compile to

```
LDA  y
LDC  10
LDC  15
ADD
STO
```

whereas

```
final int x = 10;
int   y = x + 15;
```

would compile to rather more code, as x now corresponds to a variable with storage allocated to it.

```
LDA  x
LDC  10
STO
LDA  y
LDA  x
LDV
LDC  15
ADD
STO
```

Of course, the final qualifier does allow one to define an immutable variable by a run-time expression, which is not the case for the const declaration:

```
const Max = 100;                // legal
final int TwoMax = 2 * Max;     // legal
final ThreeMax = TwoMax + Max;  // legal
const ThreeMax = 3 * Max;       // illegal
```

You might like to ponder whather the following silly code would be legal:

```
while ( i < 10) {
  final int twoI = 2 * i;
  ++i;
} // while
```

QUESTION B11 **[5 marks]**

B11 Oh dear! A rigorous test of the compiler supplied to you will reveal that compiling code like

```
void main() {                     // egB11.pav
  final set vowels  = set{'a', 'e', 'i', 'o', 'u' };
  final char ch = 'T';
  ++ch;
  vowels = vowels + set{'A', 'E', 'I'};
  vowels.Incl('O', ch);
} // main
```

will not report any errors, when of course, it should. Identify and explain the errors and modify the system to detect and report them correctly. [5 marks]

Another silly omission or two, of course! IncDecStatement needs an obvious check added:

```
    IncDecStatement                        (. DesType des;
                                              bool inc = true; .)
    =  ( "++" | "--"                       (. inc = false; .)
       ) Designator<out des>               (. if (des.entry.kind != Kinds.Var)
                                                 SemError("variable designator required");
 *                                            if (des.cannotAlter)
 *                                               SemError("you may not alter this variable");
                                              if (!IsArith(des.type))
                                                SemError("arithmetic type needed");
                                              CodeGen.IncOrDec(inc, des.type); .)
       WEAK ";" .
```

So too does AssignmentOrCall

```
    AssignmentOrCall                       (. int expType;
                                              DesType des;
                                              bool inc = true; .)
    =  (  IF (IsCall(out des))             // use resolver to handle LL(1) conflict
          identifier
          "("                              (. CodeGen.FrameHeader(); .)
             Arguments<des>                (. CodeGen.Call(des.entry.entryPoint); .)
          ")"
       |  Designator<out des>              (. if (des.entry.kind != Kinds.Var)
                                                 SemError("cannot assign to " + Kinds.kindNames[des.entry.kind
 *                                               if (des.cannotAlter)
 *                                                SemError("you may not alter this variable"); .)
          (   AssignOp
              Expression<out expType>      (. if (!Assignable(des.type, expType))
                                              SemError("incompatible types in assignment");
```

```
                                        CodeGen.Assign(des.type); .)
        |                             (. if (!IsSetType(des.type))
                                           SemError("set designator needed"); .)
          "." ( "Incl" | "Excl"       (. inc = false; .)
          )                           (. CodeGen.Dereference(); .)
          "(" ElementList<inc>
          ")"                         (. CodeGen.Pop(); .)
      )
    ) WEAK ";" .
```

QUESTION B12 **[5 marks]**

B12 You will have been taught that when an argument is passed to a function "by value", any changes made to the corresponding formal parameter should not be felt in the caller. However, if you compile and execute the following Parva code

```
void slave(set a) {              // egB12.pav
  writeLine(a);
  a.Incl(45);                    // corrupt set a
  writeLine(a);
} // slave

void main() {
  set scrap = set{30, 40, 50};
  slave(scrap);                  // pass scrap to the slave (by value)
  writeLine(scrap);
}  // main
```

the output will be

{30, 40, 50}
{30, 40, 45, 50}

{30, 40, 45, 50}

when of course it should be

{30, 40, 50}
{30, 40, 45, 50}

{30, 40, 50}

You can surely do better than that. Show us! (Hint: this can be solved very simply. Look for the very simple solution and don't get carried away!) [5 marks]

As the system was delivered, when the value of an actual set parameter is passed to a method, that parameter is an index and points to a set that is on the setHeap. This set can then become the subject of an Incl or Excl operation, which can modify that member of the setHeap directly.

The trick is to pass, not the pointer to the actual parameter, but a pointer to a new genuine copy of the actual parameter, which can be generated by using the Clone operation. A trivial fix:

```
    OneArg<Entry fp>                  (. int argType; .)
    =  Expression<out argType>        (. if (fp != null && !Assignable(fp.type, argType))
                                           SemError("argument type mismatch");
*                                        if (IsSetType(argType))
*                                          CodeGen.SetClone(); .) .
```

QUESTION B13 **[24 marks]**

B13 In the system as so far developed, there is only one set type, the elements of which are limited to non-negative integers. A more rigorous system might wish to distinguish between sets of characters and sets of integers, as suggested by the following code:

```
void main () {                     // egB13.pav
  set     ints  = set{30, 20, 10};
  charset chars = charset {'A', 'E', 'I', 'O', 'U' };
  chars.Incl('a', 'b');            // legal
  writeLine(chars);                //  {'A', 'E', 'I', 'O', 'U', 'a', 'b' };
  chars.Incl(10);                  // illegal
  ints.Incl('a', 35);              // legal, by analogy with C# char/int
  writeLine(ints);                 //  {10, 20, 30, 35, 97};
} // main
```

Identify and make the changes that would be needed to the system to accommodate these examples. (Several more changes might be needed were you to explore this fully, which time will not permit.) [24 marks]

There are quite a number of these, each one being quite small, but all being necessary. The kit handed out was already primed so that these changes would slip in very easily (provided students had studied the material given on the previous day, where most of the exam exercises had been cunningly anticipated!). Nevertheless, this question is intended as a "discriminator" and I anticipate only the top students will make real progress with it.

Clearly "charsetType" must be added to the Types class:

```
      class Types {
      // Identifier (and expression) types
        public const int
          noType      =  0,             // The numbering is significant, as
          nullType    =  2,             // array types are denoted by these
          intType     =  4,             // numbers + 1
          boolType    =  6,
          charType    =  8,
          setType     = 10,
*         charsetType = 12,
*         voidType    = 14;
```

Some of the predicates and support functions defined in the ATG file need tweaking

```
      static bool IsSetType(int type) {
*       return type == Types.setType || type == Types.charsetType;
      } // IsBool

      static int ElementType(int setType) {
        switch (setType) {
          case Types.setType     : return Types.intType;
*         case Types.charsetType : return Types.charType;
          default                : return Types.noType;
        }
      } // ElementType

      static bool Assignable(int typeOne, int typeTwo) {
      // Returns true if a variable of typeOne may be assigned a value of typeTwo
        return    typeOne == typeTwo
               || typeOne == Types.intType && typeTwo == Types.charType
               || typeOne == Types.noType
               || typeTwo == Types.noType
*              || typeOne == Types.setType && typeTwo == Types.charsetType
               || IsArray(typeOne) && typeTwo == Types.nullType;
      } // Assignable
```

We need to incorporate the new type into the ATG production for BasicType

```
       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; .) .
```

We need to check that raw integers may not be used as elements in a characterset, needing ElementList and Element to be parameterized:

```
      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");  .)
```

ElementList is accessed from Primary where sets may be defined. This needs a small change:

```
          |                                    (. type = Types.setType; .)
  *         [ "set" | "charset"                (. type = Types.charsetType; .)
            ]                                  (. int elementType = ElementType(type);
                                                  CodeGen.ConstructSet(); .)
            "{" [ ElementList<elementType, true>
                ]
            "}"
```

as does AssignmentOrCall, where sets may be built using Incl:

```
           |                                  (. if (!IsSetType(des.type))
                                                   SemError("set designator needed"); .)
             "." ( "Incl" | "Excl"            (. inc = false; .)
             )                                (. CodeGen.Dereference(); .)
             "("
             ElementList<ElementType(des.type), inc) .)
             ")"
         )
     ) WEAK ";" .
```

To support the output of a set in character form requires a few additions

```
      WriteElement                            (. int expType;
                                                 String str; .)
      =   StringConst<out str>                (. CodeGen.WriteString(str); .)
        | Expression<out expType>             (. switch (expType) {
                                                   case Types.noType:   break;
                                                   case Types.intType:
                                                   case Types.boolType:
                                                   case Types.charType:
                                                   case Types.setType:
  *                                                case Types.charsetType:
                                                     CodeGen.Write(expType); break;
                                                   default:
                                                     SemError("cannot write this type");
                                                     break;
                                                  } .) .
```

The code generator needs tweaking

```
        public static void Write(int type) {
        // Generates code to output value of specified type from top of stack
          switch (type) {
            case Types.intType:        Emit(PVM.prni);  break;
            case Types.boolType:       Emit(PVM.prnb);  break;
            case Types.charType:       Emit(PVM.prnc);  break;
            case Types.setType:        Emit(PVM.prset); break;
  *         case Types.charsetType:    Emit(PVM.prsetc); break;
          }
        } // CodeGen.Write
```

and, of course, we must tweak the PVM by adding:

```
  *       case PVM.prsetc:         // simple display of character set
  *         if (tracing) results.Write(padding);
  *         results.Write(GetSet(Pop()).ToCharSetString());
  *         if (tracing) results.WriteLine();
  *         break;
```

The productions to the Expression hierarchy can promote to an integer set type if either factor is of integer set type:

```
    AddExp<out int type>                      (. int type2;
                                                 int op; .)
    =  MultExp<out type>
       { AddOp<out op>
         MultExp<out type2>                   (. if (IsArith(type) && IsArith(type2)) {
                                                   type = Types.intType;
                                                   CodeGen.BinaryOp(op);
                                                 }
                                                 else if (IsSetType(type) && IsSetType(type2)) {
                                                   switch (op) {
                                                     case CodeGen.add : CodeGen.BinaryOp(CodeGen.uni); break;
                                                     case CodeGen.sub : CodeGen.BinaryOp(CodeGen.dif); break;
                                                     default: SemError("invalid set operation"); break;
                                                   }
*                                                  if (type != type2) type = Types.setType;
                                                 }
                                                 else {
                                                   SemError("arithmetic operands needed");
                                                   type = Types.noType;
                                                 } .)
       } .
```

QUESTION O14 (Bonus - optional) **[3 + 5 = 8 marks]**

O14 After careful consideration of the overwhelming success of the Parva language as extended this weekend, its inventor has decided to release Parva++. However, in this language, the production for *IfStatement* is to be changed to incorporate a further key word, `fi`.

```
IfStatement     = "if" "(" Condition ")" Statement
                    [ "else" Statement ]
                   "fi" .
```

The proud inventor was overheard to say that he was tired of explaining the famous "dangling else" ambiguity to students and that this syntactic change would eliminate it once and for all.

(a) What do you understand by the "dangling else" problem? [3 marks]

(b) Analyse in some detail whether the inventor's claim is correct. [5 marks]

The dangling else problem arises from an ambiguous definition of the IfStatement in many programming languages as

```
IfStatement = "if" "(" Condition ")" Statement [ "else" Statement ]
```

This has the unfortunate effect that a construct like

```
if ( condition1 ) if (condition2 ) StatementA else StatementB
```

can, without braces { }, be parsed either to "mean" or "match" statements made more explicit as

```
if ( condition1 ) {
  if ( condition2 ) StatementA else StatementB
} else { /* empty */ }
```

or as

```
if ( condition1 ) { if ( condition2 ) StatementA else { /* empty */ } }
else StatementB
```

The inventor's claim is quite correct. Consider the grammar in conjunction with a convenient non-terminal OtherStatement that produces all the statements other than the IfStatement itself.

```
Statement   = IfStatement | OtherStatement .
```

```
IfStatement = "if" "(" Condition ")" Statement [ "else" Statement ] "fi" .
```

that is, we assume that FIRST(OtherStatement) does not include "if" so that Rule 1 is satisfied for the first of these productions.

The second production, for IfStatement, contains the nullable component for which FIRST(["else" Statement]) = "else" but for which FOLLOW (["else" Statement]) = "fi", which does not conflict with Rule 2 of the familiar LL(1) constraints (unless in some fit of madness the language has used "fi" as a possible initial terminal of OtherStatement).

QUESTION O15 (Bonus - optional) **[10 marks]**

O15 Howls of anguish will arise when Parva++ is released. All those millions of lines of code produced in the Hamilton Laboratory by generations of students will, at a stroke, be rendered syntactically incorrect.

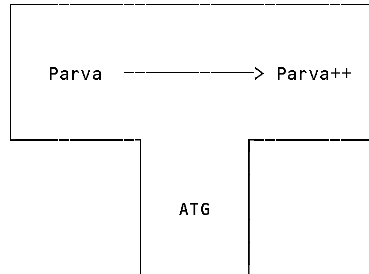
The inventor is unperturbed, and promises to use Coco/R to write a quick Parva to Parva++ translator. He argues that all he needs to do is to write a sort of pretty printer that can reformat existing Parva programs, with extra `fi` tokens inserted into *IfStatements* at the appropriate places, and sketches out a few T-diagrams to show the various steps in the process.

What do the T-diagrams look like? (Make the assumption that you have available the executable versions of the Coco/R compiler-generator for C#, an executable version of a C# compiler, the Cocol grammars for Parva and for Parva++, as well as Parva and Parva++ compilers built from these grammars, and at least one Parva program `sample.pav` to be converted to `sample.p++` and then compiled.) [10 marks]

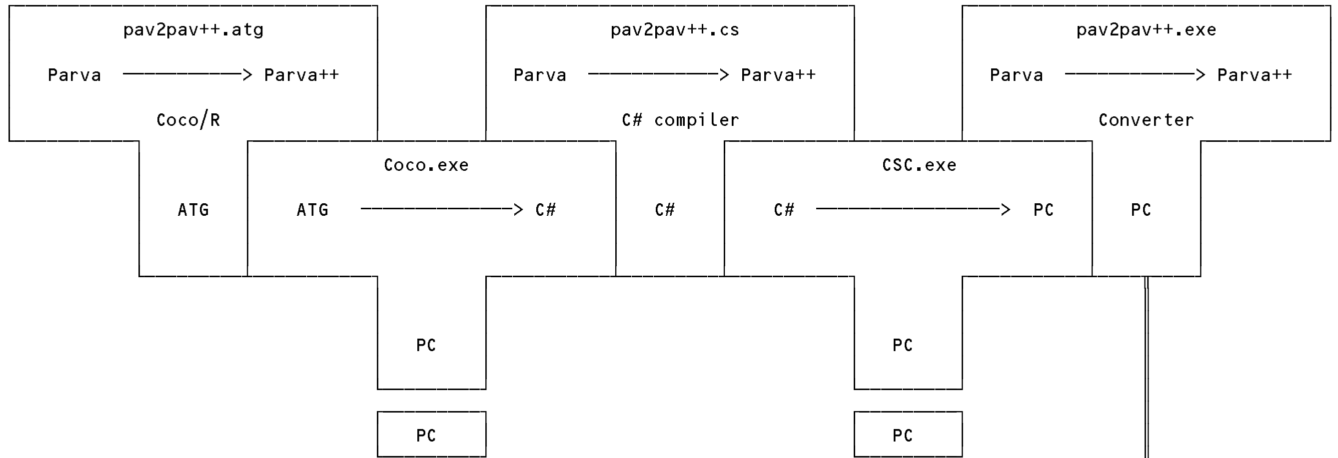
```
Develop a Cocol grammar for the high level translator ( .ATG file )

  This can use the same "front end" as
  the Parva compiler, and a very much
  simpler "back end"

  But it should produce nice readable
  Parva++ which probably means
  retaining the comments!
```



Develop the high level translator



Convert and compile sample.pav