RHODES UNIVERSITY

uSpplementary Examinations - 2017/2018

Computer Science 301 - Paper 2 - Compilers - Solutions

Examiners:	Time 4 hours
Prof P.D. Terry	Marks 180
Prof M. Kuttel	Pages 14 (please check!)

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

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

Section A: Conventional questions [100 marks]

QUESTION A1 **[2 + 4 = 6 marks]**

(a) What distinguishes a *native code compiler* from an *interpretive compiler*? [2 marks]

A native code compiler generates machine level object code, typically for the same machine that is executing the compiler itself. An interpretive compiler generates intermediate level code, typically for a virtual machine whose operation can be emulated by a suitable interpreter for such code.

(b) Suggest (with some explanation) one property of native code compilation that is claimed to offer an advantage over interpretive compilation, and also one property of interpretive compilation that is claimed to offer an advantage over native code compilation. [4 marks]

Native code compilers should produce object code that can be executed at the full speed of the host machine, typically one or two orders of magnitude faster than an interpreter can execute code for a virtual machine. Interpretive compilers on the other hand are much more easily developed, and can be made highly portable (all that is needed is a suitable interpreter for the virtual code).

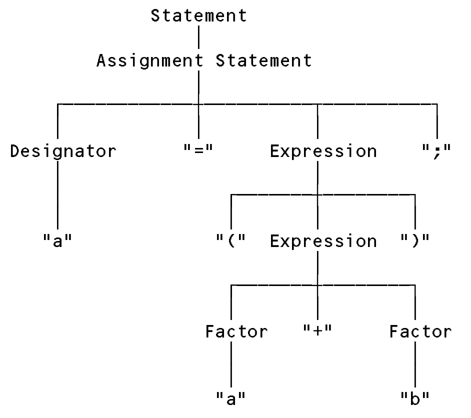
QUESTION A2 **[6 marks]**

What distinguishes a *concrete syntax tree* from an *abstract syntax tree*? Illustrate your answer by drawing both forms of tree corresponding to the simple C# statement [6 marks]

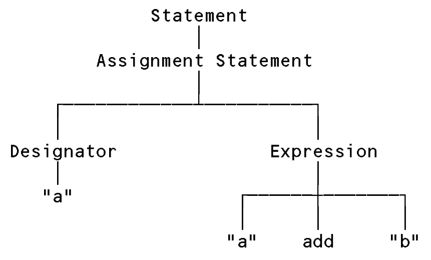
```
a = ( b + c ) ;
```

A concrete syntax tree incorporates all the tokens used in deriving the parse tree, while an abstract syntax tree retains only the information needed later to analyse the semantics completely:

Concrete



Abstract



QUESTION A3 **[6 + 10 = 16 marks]**

(a) *Scope* and *Existence/Extent* are two terms that come up in any discussion of the implementation of block-structured languages. Briefly explain what these terms mean, and the difference between them. [6 marks]

Scope is a compile-time concept - essentially it refers to the "area" of code in which an identifier can be recognized. A

Existence or Extent is a run-time concept - essentially it refers to the "real time" during which storage need be allocated to a data structure of whatever complexity.

(b) Scope rules for a simple block-structured language like Parva can be implemented by making use of a suitable data structure for the symbol table. Show what such a structure might look like when a top-down one-pass compiler reaches each of the points marked (1) and (2), if it compiles the program below. [10 marks]

```
void main () {
  int x = 10, y, z;
  while (x > 0) {
    read(y);
    bool isPosY = y > 0;
    int z = x / 2;
    x = x - y;
    // +++++++++++++++++++++++++++ point (1)
  }
  int a = x + y + z;
  // ++++++++++++++++++++++++++++++ point (2)
}
```

Solution:

Various solutions are possible, but they all rely to some extent on a stack or farm of stacks. The sort of structure exemplified in our case studies was as follows:

At point 1 we might have

```
top ────────>  isPosY ──────> z ─────────────────┐
 │                                               │
 ├──────────>  x ──────────> y ───────> z ──────┐│
 │                                              ││
 └──────────>  main  ───────────────────────────>┘└──────> sentinel (undefined) ─────>
```

At point 2 we might have

```
top ──────> x ─────> y ─────> z ───────> a ─────┐
 │                                              │
 └──────────>  main  ───────────────────────────>└──────> sentinel (undefined) ─────>
```

QUESTION A4 **[15 + 15 = 30 marks]**

(Cocol and recursive descent parsing) The following familiar Cocol code describes a set of EBNF productions (of the form found in the PRODUCTIONS section of the grammar itself).

```
COMPILER EBNF $CN
/* Describe a set of EBNF productions
   P.D. Terry, Rhodes University, 2017 */

CHARACTERS
  eol         = CHR(10) .
  space       = CHR(32) .
  control     = CHR(0) .. CHR(31) .
  letter      = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .
  digit       = "0123456789" .
  lowline     = "_" .
  printable   = ANY - control .
  nonSpace    = printable - space .
  noquote1    = ANY - "'" - control .
  noquote2    = ANY - '"' - control .

TOKENS
  nonterminal = letter { letter | lowline | digit } .
  terminal    = "'" noquote1 { noquote1 } "'" | '"' noquote2 { noquote2 } '"' .

IGNORE control

PRODUCTIONS
```

```
  EBNF       = { Production } EOF .

  Production  = nonterminal "=" Expression  "." .

  Expression  = Term { "|" Term } .

  Term        = Factor { Factor } .

  Factor      =   nonterminal | terminal | "(" Expression ")"
                | "[" Expression "]" | "{" Expression "}" .

END EBNF.
```

(a) In the original notation known as BNF, productions took a form exemplified by

```
<Children> ::= ε  |  <Child> <Children>
<Child>    ::= David | Helen | Kayd | Jamie
```

The notation allowed the use of an explicit ε. () Parentheses, [] brackets and {} braces were not used. Non-terminals and terminals were distinguished by the presence or absence of < > angle brackets, and a production was terminated at the end of a line.

Using the same CHARACTERS as in the above grammar, give the TOKENS and PRODUCTIONS sections for a Cocol grammar that uses EBNF to describe BNF style productions (don't write a BNF style grammar to descripe EBNF style productions). Use the above grammar as a guide, use `"eps"` to represent ε and use BNF as the goal symbol. [15 marks]

We have to redefine the form that terminals and non-terminals take in the TOKENS section. This is the trickiest part to get right. The transformation of the PRODUCTIONS section is straightforward. Note where the null option ε is introduced!

```
COMPILER BNF $CN
/* Describe a set of EBNF productions
  P.D. Terry, Rhodes University, 2017 */

CHARACTERS
  eol         = CHR(10) .
  space       = CHR(32) .
  control     = CHR(0) .. CHR(31) .
  letter      = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .
  digit       = "0123456789" .
  lowline     = "_" .
  printable   = ANY - control .
  nonSpace    = printable - space .
  startTerm   = printable - "<" .

TOKENS
  nonterminal = "<" letter { letter | lowline | digit | space } ">" .
  terminal    = startTerm { nonSpace } | "<"  .
  EOL         = eol .

IGNORE control - eol

PRODUCTIONS
  BNF         = { Production } EOF .
  Production  = nonterminal "::=" Expression EOL .
  Expression  = [ "ε" "|" ] Term { "|" Term } .
  Term        = Factor { Factor } .
  Factor      = nonterminal | terminal .
END BNF.
```

(b) Assume that you have available a suitable scanner method called `GetSym` that can recognize the terminals of this *new* grammar for BNF and classify them appropriately as members of the following set:

```
{ EOFSym, noSym, EOLSym, termSym, nontermSym, definedBySym, epsilonSym, barSym }
```

Develop a hand-crafted recursive descent parser for recognizing a set of such BNF productions (not the original EBNF production set given earlier), that is based on your description in (a). *(Your parser can take drastic action if an error is detected. Simply call methods like* `Accept` *and* `Abort` *to produce appropriate error messages and then terminate parsing. You are not required* *to write any code to implement the* `GetSym`*,* `Accept` *or* `Abort` *methods.)* [15 marks]

What was expected was code on the following lines:

```
static SymSet FirstFactor = new IntSet(termSym, nontermSym});

static void BNF() {
  while (sym == nontermSym) {
    Production();
  }
  Accept(EOFSym, "EOF expected");
}

static void Production() {
  GetSym();
  Accept(definedBySym, "::= expected");
  Expression();
  Accept(EOLSym, "productions must be terminated by end-of-line");
} // Production

static void Expression() {
  if (sym == epsilonSym) {
    GetSym();
    Accept(barSym, "| expected");
  }
  Term();
  while (sym == barSym) {
    GetSym(); Term();
  }
} // Expression

static void Term() {
  Factor();
  while (FirstFactor.contains(sym)) Factor();
} // Term

static void Factor () {
  switch (sym) {
    case nontermSym :
    case termSym :
      GetSym();
      break;
    default:
      Abort("invalid start to Factor");
      break;
  }
} // Factor
```

QUESTION A5 **[2 + 2 + 6 + 6 = 16 marks]**

(Grammars) By now you should be familiar with RPN or "Reverse Polish Notation" as a notation that can describe expressions without the need for parentheses. The notation eliminates parentheses by using "postfix" operators after the operands. To evaluate such expressions one uses a stack architecture, such as formed the basis of the PVM machine studied in the course. Examples of RPN expressions are:

`3 4 +`	- equivalent to	`3 + 4`
`3 4 5 + *`	- equivalent to	`3 * (4 + 5)`

In many cases an operator is taken to be "binary" - applied to the two preceding operands - but the notation is sometimes extended to incorporate "unary" operators - applied to one preceding operand:

`4 sqrt`	- equivalent to	`sqrt(4)`
`5 -`	- equivalent to	`-5`

Here are two attempts to write grammars describing an RPN expression:

```
(G1)    RPN      =    RPN RPN binOp
                    | RPN unaryOp
                    | number .
        binOp    =  "+" | "-" | "*" | "/" .
        unaryOp  =  "-" | "sqrt" .
```

and

```
(G2)    RPN       =   number REST .
        REST      =   [ number REST binOp REST | unaryOp ].
        binOp     =   "+" | "-" | "*" | "/" .
        unaryOp   =   "-" | "sqrt" .
```

(a) Give a *precise* definition of what you understand by the concept of an *ambiguous grammar*. [2 marks]

An ambiguous grammar is one for which at least one sentence can be derived in more than one way, that is, for which a parse tree is not unique.

(b) Give a *precise* definition of what you understand by the concept of *equivalent grammars*. [2 marks]

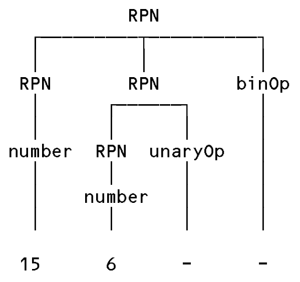
Grammars are equivalent if they derive exactly the same set of sentences (not necessarily using the same set of sentential forms or parse trees).

(c) Using the expression

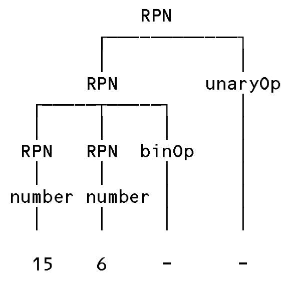
```
15  6  -  -
```

as an example, and by drawing appropriate parse trees, demonstrate that both of the grammars above are ambiguous. [6 marks]

Using grammar G1:

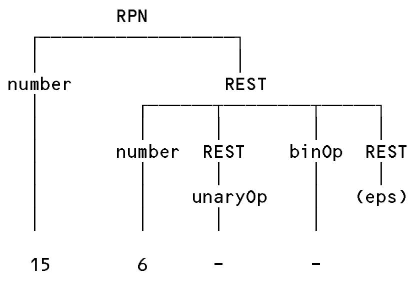


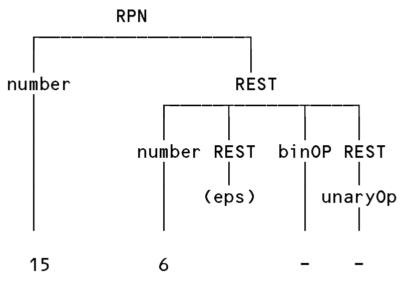
(implied value is 21)



(implied value is -9)

Using grammar G2:





(d) Analyse each of these grammars to check whether they conform to the LL(1) conditions, explaining quite clearly (if they do not!) where the rules are broken. [6 marks]

G1 is left recursive and thus cannot be an LL(1) grammar. There are two alternatives for the right side of the production for RPN that both start with RPN, so Rule 1 is broken

*For G2, REST is nullable. First(REST) is { number, sqrt, - } while Follow(REST) is { +, -, /, * } so Rule 2 is broken.*

QUESTION A6 **[2 + 4 + 14 = 20 marks]**

(a) What distinguishes a *context free grammar* from a *context sensitive grammar*? [2 marks]

In general, grammar productions take the form

$$\alpha \rightarrow \beta \qquad \text{with} \quad \alpha \in (N \cup T)^* N (N \cup T)^* \; ; \; \beta \in (N \cup T)^*$$

where N is the set of non-terminals and T is the set of terminals. In context-free grammars $\alpha \in N$ (the left side of each production must consist of a single non-terminal); for a grammar to be context-sensitive there must be at least one production for which α is comprised of two or more tokens. Note the importance of the phrase "at least one production".

(b) The syntax of many programming languages is described by a context free grammar, and yet there are properties of most programming languages that are context sensitive. Mention one such property, and indicate briefly how this context sensitivity is handled in practical compilers. [4 marks]

There are plenty of examples to choose from - such as

- *variables must be declared before they are used*
- *the number of formal and actual arguments for functions must agree*
- *expressions used to control if and while statements must be of Boolean type*
- *values assigned to variables must be of the appropriate type*
- *break statements may only be used in the context of a loop or switch statement*
- *a nice one suggested by (c) below - in a multiple assignment statement there must be as many "designators" as there are "expressions".*

Many of these are usually handled by making use of a "symbol table" in which the various properties of the items denoted by identifiers are recorded, or by checking one count against another.

(c) *(Static semantics)* A BYTE (Bright Young Terribly Eager) student has been nibbling away at writing extensions to her first Parva compiler, while learning the Python language at the same time. She has been impressed by a Python feature that allows one to write multiple assignments into a single statement, as exemplified by

```
A, B = X + Y, 3 * List[6];
A, B = B, A;                 // exchange A and B
A, B = X + Y, 3, 5;          // incorrect
```

which she correctly realises can be described by the context-free production

```
Assignment = Designator { "," Designator } "=" Expression { "," Expression } ";"
```

So far the students have not been shown how to generate code, but have been made aware of the need for static semantic checking, as exemplified in the following simple production for an *Assignment* (for very simple assignments only):

```
Assignment                      (. DesType des;
                                   int type; .)
=  Designator<out des>          (. if (des.entry.kind != Kinds.Var)
                                     SemError("invalid assignment"); .)
   "=" Expression<out int type>
   ";" .
```

Assuming that her Parva compiler will eventually support Boolean, integer and character types, what static semantic checks will be needed to implement the Python-like language extension, and will these entail any special techniques? The Cocol and C# code that is needed turns out to be quite intricate, so you need not attempt to give it in detail unless you wish to do so. Give an answer on the lines of

> Each of the Designators must correspond to a variable (not a function or constant)
> ....

[14 marks]

The insight required is that all Designators must correspond to variables (or array elements); there must be as many Designators as there are Expressions (necessitating a counting system); each of the expressions must be assignment compatible with the corresponding Designator (necessitating the use of a dynamically constructed list of Designator type information). Bonus for anyone with the insight that if the numbers of Designators and Expressions disagree (and with more Expressions than Designators), then care must be taken not to try to access a non-existent member of the list of Designator types.

```
Assignment                      (. int expType;
                                   DesType des;
                                   List<DesType> desList = new List<DesType>(); .)
=  Designator<out des>          (. if (des.entry.kind != Kinds.Var)
                                     SemError("invalid assignment");
                                   desList.Add(des); .)
    { "," Designator<out des>   (. if (des.entry.kind != Kinds.Var)
                                     SemError("invalid assignment");
                                   desList.Add(des);

    }
    AssignOp                    (. int count = 0, expCount = 1; .)
    Expression<out expType>     (. if (count < desList.Count
                                       && !Assignable(desList[count++].type, expType))
                                     SemError("incompatible types in assignment"); .)
    { ","
      Expression<out expType>   (. if (count < desList.Count
                                       && !Assignable(desList[count++].type, expType))
                                     SemError("incompatible types in assignment");
                                   expCount++; .)
    }                           (. if (expCount != desList.Count)
                                     SemError("left and right counts disagree"); .)
WEAK ";"  .
```

QUESTION A7 **[3 + 3 = 6 marks]**

In the compiler studied in the course the following production was used to handle code generation for a *WhileStatement*:

```
WhileStatement<StackFrame frame>      (. Label startLoop = new Label(known); .)
=  "while" "(" Condition ")"          (. Label loopExit = new Label(!known);
                                         CodeGen.BranchFalse(loopExit); .)
   Statement<frame>                   (. CodeGen.Branch(startLoop);
                                         loopExit.Here(); .) .
```

This generates code that matches the template

```
startLoop:    Condition
              BZE  loopExit
              Statement
              BRN  startLoop
loopExit:
```

Some authors contend that it would be preferable to generate code that matches the template

```
whileLabel:   BRN  testLabel
loopLabel:    Statement
testLabel:    Condition
              BNZ  loopLabel
loopExit:
```

Their argument is that in most situations a loop body *is* executed many times, and that the efficiency of the system will be markedly improved by executing only one conditional branch instruction on each iteration.

(a) Do you think the claim is justified for an interpreted system such as we have used? Explain your reasoning. [3 marks]

While this might be the case on some pipelined architectures, on the interpreted system used in the course it would probably make very little difference. Far more time is likely to be spent executing the many statements that make up the loop body than in executing an extra unconditional branch instruction.

(b) If the suggestion is easily implementable in terms of our code generating functions, show how this could be done. If it is not easily implementable, why not? [3 marks]

It would be hard to implement using the simple minded code generator the students saw in this course, as we should need to delay generating the code for `Condition` *until after we had generated the code for a* `Block`*. Generating the various branch instructions would be easy enough, of course. In a more sophisticated compiler that built a tree representation of the program before emitting any code, judicious tree walking would allow one to get the effect relatively simply.*

Section B [80 marks]

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

Until yesterday, the Parva language that you have got to know so well had but three basic data types - integer, boolean, and character. Although it made provision for the use of string constants in `read` and `write` statement lists, it did not provide for the declaration and use of string variables:

```
void main() { $C+
  string yourName, myName, theBoff;
  myName = "Pat";
  theBoff = myName;
  read("What is your name? ", yourName);
  write(myName, " is pleased to meet ", yourName);
  if (myName != yourName)
    writeLine(" (Our names are different)");
  if (upper(myName) == upper(yourName))
    writeLine("(Our uppercased names are the same)");
} // main
```

Yesterday you made history when you were invited to develop a version of the Parva compiler that at last incorporated a `string` type, one which should have been able to support the following (at least):

(a) Declare variables and arrays of type `string`;
(b) Assign constant strings to such variables, and values of string variables to other string variables;
(c) Read and write values for string variables;
(d) Compare two strings for equality or inequality;
(e) Provide a function that returns the UPPERCASE version of a string;
(f) Perform any necessary semantic and constraint checking on strings.

Later in the day you were provided with a sample solution to that challenge. Continue now to answer the following unseen questions. **Where possible, express your answers in code rather than loose discussion.**

QUESTION B8 **[5 marks]**

The compiler as provided does not allow you to declare strings as named constants. Show (preferably give code) how this might be done, to allow, for example: [5 marks]

```
void main () {
  const title = "The thin edge of the wedge";
  string s = title;
} // main
```

This is very easy - and in fact makes for a simpler system than the original. We change the Constant parser

```
Constant<out ConstRec con>          (. con = new ConstRec();
                                       string str; .)
=   IntConst<out con.value>         (. con.type  = Types.intType; .)
  | CharConst<out con.value>        (. con.type  = Types.charType; .)
  | StringConst<out str>            (. con.type  = Types.strType;
                                       con.value = CodeGen.GetStringIndex(str); .)  /* +++++++++++++++ */
  | "true"                          (. con.type  = Types.boolType; con.value = 1; .)
  | "false"                         (. con.type  = Types.boolType; con.value = 0; .)
  | "null"                          (. con.type  = Types.nullType; con.value = 0; .) .
```

and delete the alternative for StringConst that was previously in Primary:

```
  | StringConst<out str>            (. CodeGen.LoadConstant(CodeGen.GetStringIndex(str));
                                       type = Types.strType; .)
```

QUESTION B9 **[10 marks]**

String handling libraries usually make provision for determining the length of a string. Show how to extend the compiler, code generator and interpreter to allow for Parva code like

```
void main () {
  string str;
  read(str);
  write(str, length(str) );
} // main
```

where the `length(str)` function can only be applied to a single parameter expression of `string` type. [10 marks]

This is very easy. We add a further option to Primary:

```
| "length" "("
      Expression<out type>          (. if (type != Types.strType)
                                         SemError("string parameter expected");
                                       CodeGen.Length();
                                       type = Types.intType; .)
  ")"
```

where the code generation is as follows:

```
public static void Length() {
// Generates code to determine the length of the string at tos
  Emit(PVM.leng);
} // CodeGen.Length
```

and the interpretation is simply

```
case PVM.leng:
  Push(GetString(Pop()).Length);
  break;
```

Although not shown here, PVM.leng has to be added to the enumeration of the opcodes, and a suitable mnemonic added to represent it for the purposes of listing code.

QUESTION B10 **[10 marks**

What does the system currently output if a program like the following is compiled and run?

```
void main () {
  string str;  // declared
  write(str);  // but never properly defined
} // main
```

Explain your reasoning. [10 marks]

It prints out "undefined string". The string pool is initialised to have such a string as the zero-th entry. Since the variables in the stack frame are all initialised to zero, the variable str will point to this entry. This is probably not immediately obvious; readers will have to look at quite a bit of code to find the answer.

QUESTION B11 **[15 marks]**

The situation in B10 is actually symptomatic of a larger problem. Some implementations of languages ensure that all variables not explicitly "initialised" at the point of creation are implicitly initialised to a known value - say zero - anyway. How could your compiler be changed to incorporate this device - both for simple variables and also for arrays (preferably give code)? [15 marks]

It might be argued that it does not matter, as the entire simulated memory is loaded with zero before compilation commences. However, to be safe, we should use something like this:

For simple variables we would need to change OneVar:

```
OneVar<StackFrame frame, int type>  (. int expType; .)
=                                   (. Entry var = new Entry(); .)
   Ident<out var.name>              (. var.kind = Kinds.Var;
                                       var.type = type;
                                       var.offset = frame.size;
                                       frame.size++;
                                       CodeGen.LoadAddress(var); .)     /* ++++++++++++++ */
   (   AssignOp
       Expression<out expType>      (. if (!assignable(var.type, expType))
                                         SemError("incompatible types in assignment"); .)
     |  /* force initialisation */  (. CodeGen.LoadConstant(0); .)      /* ++++++++++++++ */
   )                                (. CodeGen.Assign(var.type);        /* ++++++++++++++ */
                                       Table.Insert(var); .)
.
```

For arrays we would change the interpretation of PVM.anew. I expect few will see this!

```
case PVM.anew:          // heap array allocation
  int size = Pop();
  if (size <= 0 || size + 1 > cpu.sp - cpu.hp - 2)
    ps = badAll;
  else {
    mem[cpu.hp] = size;
    Push(cpu.hp);
    for (loop = 1; loop <= size; loop++) mem[cpu.hp + loop] = 0;          /* ++++++++++++++ */
    cpu.hp += size + 1;
  }
  break;
```

QUESTION B12 **[5 marks]**

Show how the system could be extended to report situations like that in B10 as runtime errors. [5 marks]

Change the PVM.GetString method to

```
public static string GetString(int i) {
// Retrieve string i from the string pool
  if (i == 0) ps = nullRef;                                               /* +++++++++++++ */
  return strings[i];
} // PVM.GetString
```

This will have the effect of catching errors in the use of length, concat and so on as well. Unfortunately it would still allow one to assign undefined strings to other variables, unless a more sophisticated version of PVM.sto was developed. The details of this are left as an interesting exercise!

QUESTION B13 **[20 marks]**

The compiler currently allows for string comparisons for equality and inequality. Show the modifications that are needed to support "ordering" comparisons, such as exemplified by [20 marks]

```
void main () {
  string s1, s2;  // declared
  read(s1, s2);   // initialised
  if (s1 < s2)    // compare
    write(s1, " is lexically less than ", s2);
} // main
```

We need to modify the RelExp production

```
RelExp<out int type>              (. int type2;
                                     int op; .)
=  AddExp<out type>
   [ RelOp<out op>
     AddExp<out type2>            (. if (!Comparable(type, type2))          /* +++++++++++++ */
                                       SemError("incomparable operands");
                                     CodeGen.Comparison(op, type); type = Types.boolType; .)
   ] .
```

and may conveniently introduce another Boolean "helper" method that returns true if two values may be compared for relative ordering:

```
static bool Comparable(int typeOne, int typeTwo) {
// Returns true if typeOne may be compared relative to typeTwo
  return    IsArith(typeOne) && IsArith(typeTwo)
         || typeOne == Types.strType && typeTwo == Types.strType;
} // Comparable
```

Code generation can be achieved by an obvious extension to the comparison method:

```
public static void Comparison(int op, int type) {
// Generates code to pop two values A,B of comparable type from evaluation stack
// and push Boolean value A op B
  if (type == Types.strType)
    switch (op) {
      case CodeGen.ceq:  Emit(PVM.ceqs); break;
      case CodeGen.cne:  Emit(PVM.cnes); break;
      case CodeGen.clt:  Emit(PVM.clts); break;  /* ++++++++++++++++ */
      case CodeGen.cle:  Emit(PVM.cles); break;  /* ++++++++++++++++ */
      case CodeGen.cgt:  Emit(PVM.cgts); break;  /* ++++++++++++++++ */
      case CodeGen.cge:  Emit(PVM.cges); break;  /* ++++++++++++++++ */
      default: Parser.SemError("Compiler error - bad operator"); break;
    }
  else
    switch (op) {
      case CodeGen.ceq:  Emit(PVM.ceq);  break;
      case CodeGen.cne:  Emit(PVM.cne);  break;
      case CodeGen.clt:  Emit(PVM.clt);  break;
      case CodeGen.cle:  Emit(PVM.cle);  break;
      case CodeGen.cgt:  Emit(PVM.cgt);  break;
      case CodeGen.cge:  Emit(PVM.cge);  break;
      default: Parser.SemError("Compiler error - bad operator"); break;
    }
} // CodeGen.Comparison
```

and interpretation follows as a variation on that for the arithmetic comparisons, where we have to dip into the string heap:

```
case PVM.clts:           // logical less (strings)
  tos = Pop(); Push(String.Compare(GetString(Pop()), GetString(tos)) < 0 ? 1 : 0);
  break;
case PVM.cles:           // logical less or equal (strings)
  tos = Pop(); Push(String.Compare(GetString(Pop()), GetString(tos)) <= 0 ? 1 : 0);
  break;
case PVM.cgts:           // logical greater (strings)
  tos = Pop(); Push(String.Compare(GetString(Pop()), GetString(tos)) > 0 ? 1 : 0);
  break;
case PVM.cges:           // logical greater or equal (strings)
  tos = Pop(); Push(String.Compare(GetString(Pop()), GetString(tos)) >= 0 ? 1 : 0);
  break;
```

QUESTION B14 **[15 marks]**

String concatenation (joining) is another useful feature. Show how the compiler, code generator and interpreter should be modified to allow for a program like [15 marks]

```
void main () {
  string s1, s2;  // declared
  s1 = "start ";
  s2 = "end";     // initialised
  string joined = s1 + s2;
  write(joined);  // start end
} // main
```

Overloading the + sign require awkward modification to the AddExp parser, to deal with + as a special case:

```
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 (type == Types.strType &&      /* +++++++++++++ */
                                              type2 == Types.strType &&
                                              op == CodeGen.add)
                                       CodeGen.Concat();
                                     else {                                 /* +++++++++++++ */
                                       SemError("arithmetic operands needed");
                                       type = Types.noType;
                                     } .)
   } .
```

code generation is handled by

```
public static void Concat() {
// Generates code to return the concatenation of the strings at tos and sos
  emit(PVM.concat);
} // CodeGen.Concat
```

and interpretation by

```
case PVM.concat:
  tos = Pop();
  Push(AddString(String.Concat(GetString(Pop()), GetString(tos))));
  break;
```

END OF THE EXAMINATION QUESTIONS

Section C

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

Candidates were provided with the basic ideas, and were invited to extend a version of the Parva compiler to incorporate a `string` type.

It was pointed out that string types in languages that support them usually come with a complete support library of useful functions, but to begin with they might limit themselves to a system that would do the following:

(a) Declare variables and arrays of type `string`;
(b) Assign constant strings to such variables, and values of string variables to other string variables;
(c) Read and write values for strings;
(d) Compare two strings for equality or inequality;
(e) Provide a function that returns an UPPERCASE version of a string.

They were provided with an exam kit, containing the Coco/R system, a working Parva compiler that handles integer, character and boolean types, along with a suite of simple, suggestive test programs. They were told that later in the day some further ideas and hints would be provided.

Section D

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

A complete Parva compiler, incorporating one approach to the basic provision of a simple `string` type, was supplied to candidates in a later version of the examination kit. They were encouraged to study it in depth and warned that questions in Section B would probe this understanding; few hints were given as to what to expect, other than that they might be called on to comment on the solution, and perhaps to make some modifications and extensions. They were also encouraaged to spend some time thinking how any other ideas they had during the earlier part of the day would need modification to fit in with the solution kit presented to them.