RHODES UNIVERSITY

November Examinations - 2007

Computer Science 301 - Paper 1 - Solutions

Examiners:
Prof P.D. Terry
Prof S. Berman

Time 3 hours
Marks 180
Pages 11 (please check!)

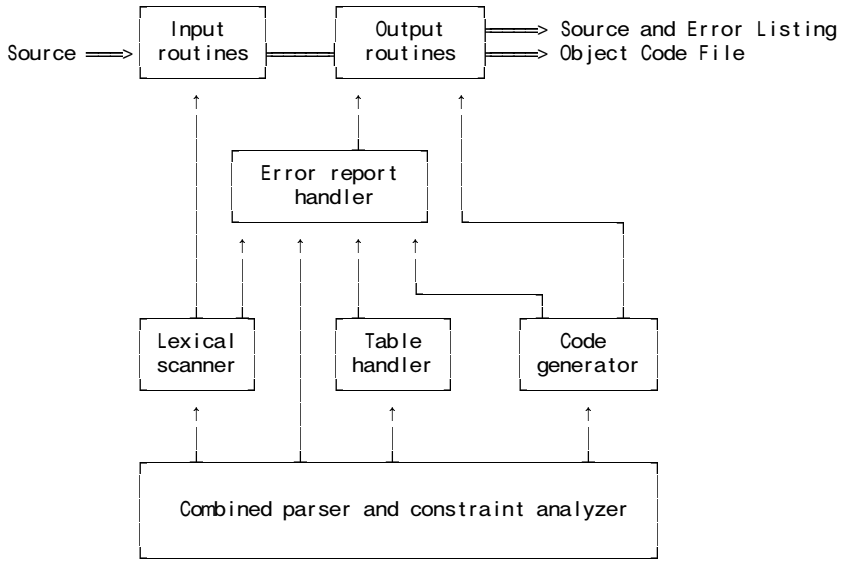
Answer all questions. Answers may be written in any medium except red ink.

(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 develop a Parva compiler that could handle aspects of a `string` *type. 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 [100 marks]

A1. *(Compiler structure)* A syntax-directed compiler usually incorporates various components, of which the most important are the scanner, parser, constraint analyser, error reporter, code generator, symbol table handler and I/O routines. Draw a diagram indicating the dependence of these components on one another, and in particular the dependence of the central syntax analyser on the other components. Also indicate which components constitute the *front end* and which the *back end* of the compiler. [10 marks]

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



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

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

```
COMPILER Index $CN
/* Grammar describing index in a book
   P.D. Terry, Rhodes University, 2007 */

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

```
TOKENS
/* Notice the careful and unusual definition for word */
  word       = startword { inword } .
  number     = digit { digit } .
  EOL        = eol .

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

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

Assume that you have available a suitable scanner method called `getSym` that can recognize the terminals of *Index* and classify them appropriately as members of the following enumeration

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

Develop a hand-crafted recursive descent parser for recognizing the index of a book based on the grammar above. *(Your parser can take drastic action if an error is detected. Simply call methods like* `accept` *and* `abort` *to produce appropriate error messages and then terminate parsing. You are not required to write any code to implement the* `getSym`*,* `accept` *or* `abort` *methods.)* [16 marks]

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

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

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

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

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

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

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

A3. *(Grammars)* Formally, a grammar G is defined by a quadruple $\{ N, T, S, P \}$ with the four components

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

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

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

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

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

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

(1) FIRST(σ) where $\sigma \in (N \cup T)^+$

a FIRST(σ) if $\sigma \Rightarrow^* a\tau$ (a T ; $\sigma, \tau \in (N \cup T)^*$)

(2) FOLLOW(A) where A ε N

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

(3) A context-free grammar

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

(4) A reduced grammar

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

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

for some strings α and β, and where

$$B \Rightarrow^* \gamma$$

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

(b) In terms of the notation here, concisely state the two rules that must be satisfied by the productions of a context free grammar in order for it to be classified as an LL(1) grammar. [6 marks]

Rule 1

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

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

the sets of initial terminal symbols of all strings that can be generated from each of the alternative ξ_{ik} must be disjoint, that is

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

Rule 2

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

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

but where $\xi_{ik} \Rightarrow \varepsilon$ for some k, the sets of initial terminal symbols of all sentences that can be generated from each of the ξ_{ij} for $j \neq k$ must be disjoint from the set FOLLOW(A_i) of symbols that may follow any sequence generated from A_i, that is

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

or, rather more loosely,

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

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

$S \rightarrow A\ B$
$A \rightarrow a\ A \mid a$
$B \rightarrow b\ B\ c \mid bc$

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

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

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

$S \rightarrow A\ B$
$A \rightarrow a\ \{ a \}$
$B \rightarrow b\ \{ B \}\ c$

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

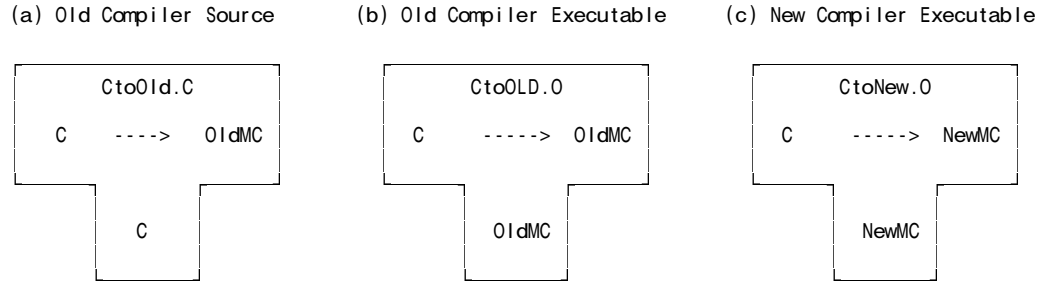
$S \rightarrow B\ .$
$B \rightarrow a\ B\ b\ B \mid b\ B\ a\ B \mid \varepsilon$

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

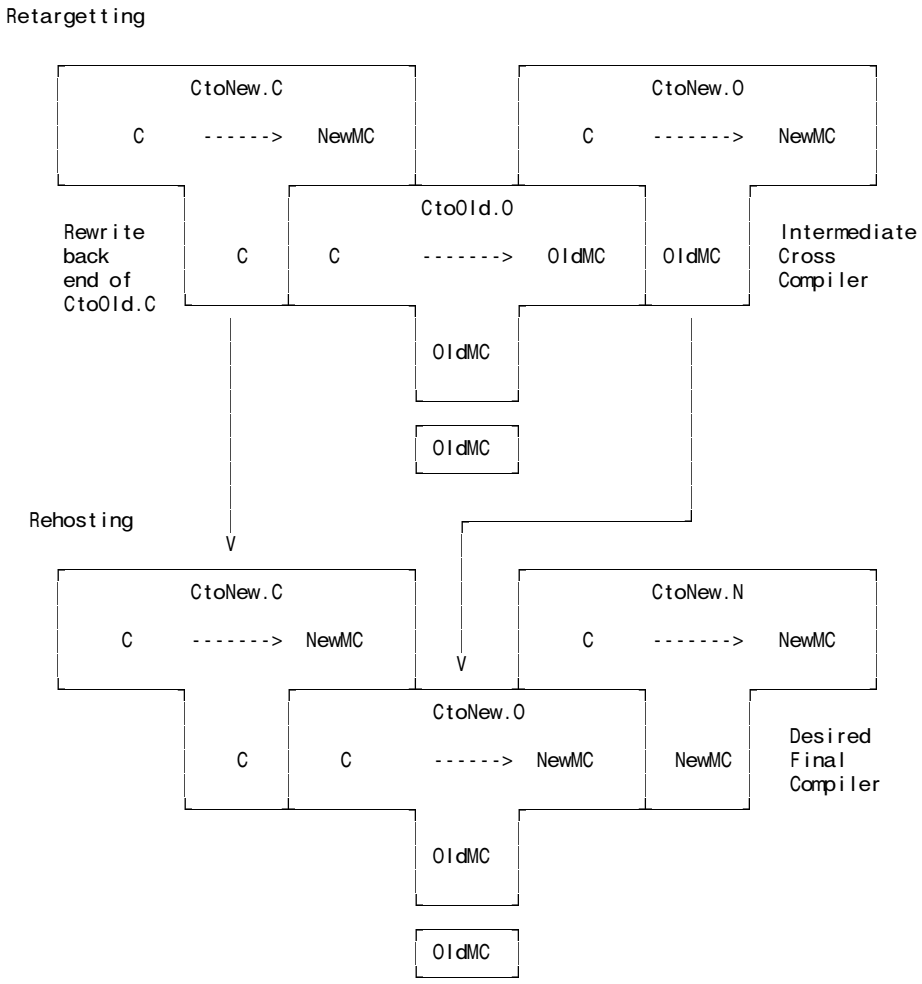
(f) "Keep it as simple as you can, but no simpler" said Einstein. Strings that might be members of the language of (e) can surely be accepted or rejected by a very simple algorithm. Give such an algorithm, using a Java-like notation. [3 marks]

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

A4. *(T diagrams)* The process of "porting" a compiler to a new computer incorporates a *retargetting* phase (modifying the compiler to produce target code for the new machine) and a *rehosting* phase (modifying the compiler to run on the new machine). Illustrate these two phases for porting a C compiler, by drawing a set of T diagrams. Assume that you have available the compilers (a) and (b) below and wish to produce compiler (c). [16 marks]



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



A5. *(Attributed Grammars in Cocol)* XML (eXtensible Markup Language) is a powerful notation for marking up data documents in a portable way. XML code looks rather like HTML code, but has no predefined tags. Instead a user can create customized markup tags, similar to those shown in the following extract.

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

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

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

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

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

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

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

Show how your grammar should be attributed to perform such checks. [18 marks]

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

```
COMPILER XML $CN
/* Parse a set of simple XML elements (no attributes)
   P.D. Terry, Rhodes University, 2007 */

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

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

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

```
IGNORE  CHR(0) .. CHR(31)

PRODUCTIONS
   XML     = Element  { Element } .

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

END XML.
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Section B [80 marks]

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

It was in 1988, at the second of a series of conferences held by a group charged with developing a rigorous standard for the definition of Modula-2, that one Susan Eisenbach made a telling observation: "Modula-2 would be the perfect language if we could add just one more feature. The difficulty is that nobody can decide what that one feature should be". She was right. Language designers cannot resist the temptation to add extension after extension to their brainchild.

So it is with Parva.

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:

```
void main() { $C+
  int i;
  read("Supply a value for i" , i);
  write("The value you supplied was ", i);
}
```

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)");
}
```

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 do 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 strings
(d) Compare two strings for equality or inequality
(e) Provide a function for converting a string to UPPERCASE.
(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.

B7. The compiler as provided does not allow you to declare strings as named constants. Suggest how this might be done, to allow, for example: [5 marks]

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

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  = Entry.intType; .)
  | CharConst<out con.value>            (. con.type  = Entry.charType; .)
  | StringConst<out str>                (. con.type  = Entry.strType;
                                           con.value = CodeGen.getStringTop(str); .)  /* ++++++++++++++++ */
  | "true"                              (. con.type  = Entry.boolType; con.value = 1; .)
  | "false"                             (. con.type  = Entry.boolType; con.value = 0; .)
  | "null"                              (. con.type  = Entry.nullType; con.value = 0; .) .
```

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

```
| StringConst<out str>                (. CodeGen.loadConstant(CodeGen.getStringTop(str));
                                         type = Entry.strType; .)
```

B8. String handling libraries usually make provision for determining the length of a string. How would the compiler, code generator and interpreter have to be extended to allow for code like

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

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 != Entry.strType)
                                         SemError("string parameter expected");
                                       CodeGen.length();
                                       type = Entry.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);
}
```

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.

B9. 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
}
```

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.

B10. The situation in B9 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)? [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 = Entry.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 could change the interpretation of PVM.anew. Only one person saw 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;
```

B11. How could the system be changed to report situations like that in B9 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 (String) strings.get(i);
}
```

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!

B12. The compiler currently allows for string comparisons for equality and inequality. How would it need to be modified to allow for "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);
}
```

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 = Entry.boolType; .)
   ] .
```

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

```
static boolean comparable(int typeOne, int typeTwo) {
// returns true if typeOne may be compared relative to typeTwo
  return    isArith(typeOne) && isArith(typeTwo)
         || typeOne == Entry.strType && typeTwo == Entry.strType;
}
```

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 == Entry.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;
      }
    }
```

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

```
case PVM.clts:           // logical less (strings)
  tos = pop(); push(getString(pop()).compareTo(getString(tos)) < 0 ? 1 : 0);
  break;
case PVM.cles:           // logical less or equal (strings)
  tos = pop(); push(getString(pop()).compareTo(getString(tos)) <= 0 ? 1 : 0);
  break;
case PVM.cgts:           // logical greater (strings)
  tos = pop(); push(getString(pop()).compareTo(getString(tos)) > 0 ? 1 : 0);
  break;
case PVM.cges:           // logical greater or equal (strings)
  tos = pop(); push(getString(pop()).compareTo(getString(tos)) >= 0 ? 1 : 0);
  break;
```

B13. String concatenation (joining) is another useful feature. How would your compiler, code generator and interpreter have to be modified to allow for a program like [15 marks]

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

Overloading the + sign require slightly 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 = Entry.intType;
                                         CodeGen.binaryOp(op);
                                       }
                                       else if (type == Entry.strType &&      /* ++++++++++++++ */
                                                type2 == Entry.strType &&
                                                op == CodeGen.add)
                                         CodeGen.concat();
                                       else {                                 /* ++++++++++++++ */
                                         SemError("arithmetic operands needed");
                                         type = Entry.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);
}
```

and interpretation by

```
case PVM.concat:
  tos = pop();
  push(addString(getString(pop()).concat(getString(tos))));
  break;
```

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 string literals 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 for converting a string to UPPERCASE
(f) Perform any necessary semantic and constraint checking on strings.

They were provided with an exam kit for Java or C#, containing a working compiler for integer 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.

Free information

Listings were provided of the following:

Summary of useful library classes

Strings and Characters in Java

Simple list handling in Java

Simple list handling in C#