RHODES UNIVERSITY

November Examinations - 2005

Computer Science 301 - Paper 2 - solutions

Examiners:
Prof P.D. Terry
Dr J.H. Greyling

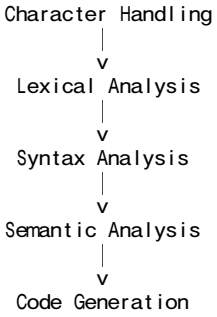
Time 3 hours
Marks 180
Pages 11 (please check!)

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 the full text of Section B, summaries of useful Java library classes, and a Parva grammar. During the examination, candidates were given machine executable versions of the Coco/R compiler generator, access to a computer and machine readable copies of the questions.)

Section A [90 marks]

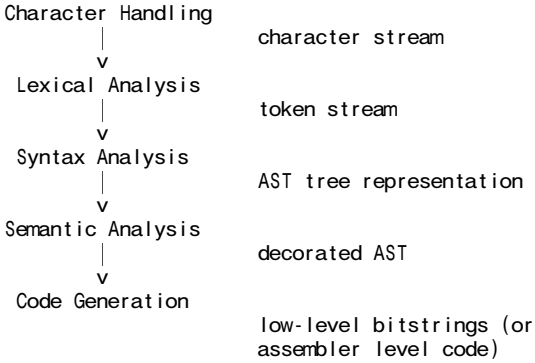
A1. A compiler is often described as incorporating several "phases", as shown in the diagram below



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

(a) Annotate the diagram to suggest the changes that take place to the representation of a program as it is transformed from source code to object code by the compilation process. [4 marks]

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.



(b) Suggest - perhaps by example - the sort of errors that might be detected in each of the phases of the compilation process. [4 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 has seen, other code generation errors could arise from trying to generate too much code for the simulated memory of the virtual machine to handle.

A2. (a) The regular expression 1 (1 | 0)* 0 describes the binary representations of the non-zero even integers (2, 4, 6, 8 …). Give a regular expression that describes the binary representations of non-negative integers that are exactly divisible by 4 (four), that is (0, 4, 8, 12, 16 …) [3 marks].

divisibleBy4 = 0 | 1 (1 | 0) 0 0*

Most people do not notice that one needs a special case to handle "zero". "1 (1 | 0) 0 0" only caters for the non-zero values!*

(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. Either complete the Cocol description of a token that would be recognized as a currency value or, equivalently, use conventional regular expression notation to describe a currency value. [5 marks]

Your solution should not accept a representation that has leading zeroes, like R001,123.00.

non-zero = 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
digit = 0 | non-zero
currency = R (0 | non-zero (digit | ε) (digit | ε) (, digit digit digit)) . digit digit*

```
CHARACTERS
   nonzero = "123456789" .
   digit   = nonzero + "0" .

TOKENS
   currency = "R" ( "0" | nonzero [ digit [ digit ]] { "," digit digit digit } )
              "." digit digit .
```

A3. Consider the following simple grammar. It has one nonterminal A and two terminals (and).

A = "(" A ")" A | ε . (G1)

(a) Is G1 an LL(1) grammar? If not, why not? [4 marks]

The production for A has two alternatives. The FIRST sets for these two alternatives are { "(" } and the empty set Φ, so Rule 1 is trivially satisfied. Since A is nullable we must test Rule 2 (we only need do this if a non-terminal is nullable; people miss this point). To determine FOLLOW(A) look at the right hand sides of the productions for instances of A. This only happens in the situation where A is followed by ")", so FOLLOW(A) = { ")" } and so Rule 2 is satisfied, and the grammar is LL(1). Many people might wrongly dream up the idea that FOLLOW(A) = { "(" , ")" }

(b) What language does G1 describe? (Write down a few strings derived from A and the pattern should be obvious.) [3 marks]

It's the language of "matched parentheses". Strings of nested parentheses are permitted, such as ((( ))), but so are strings like ( ) ( ) ( ), and of course combinations of these like ((( ))) ( ) (( ) ( )). Cute, really - amazing what a simple grammar can do. Note that the "language" cannot contain any As in the strings it produces; these can only appear in the intermediate sentential forms.

(c) Is G1 an ambiguous grammar? Explain your reasoning. [2 marks]

It cannot be an ambiguous grammar - we have shown it to be an LL(1) grammar, and hence it must be both deterministic and non-ambiguous.

Now consider the following grammar that looks rather similar but does not have an ε:

A = "(" A ")" | "(" ")" | A A . (G2)

(d) Is G2 an LL(1) grammar? If not, why not? [2 marks]

This one is not LL(1). There are three alternatives for A, namely "(" A ")" and "(" ")" and AA. All three alternatives have the same FIRST set, namely { "(" } so Rule 1 is broken.

(e) What language does G2 describe? (Write down a few strings derived from A and the pattern should be obvious.) [3 marks]

It's also a language of "matched parentheses" - the two grammars are almost equivalent in that sense.
But G1 allows a completely empty string, while in G2 the shortest allowed string is () .
will be interesting to see whether people spot this!

(f) Is G2 an ambiguous grammar? Explain your reasoning. [2 marks]

Students tend to fall into the trap of saying "if a grammar is non-LL(1), it must be ambiguous". That is not correct, though the opposite deduction, namely that "if a grammar is LL(1), it cannot be ambiguous", always holds (funny stuff, logic!). The class had seen several examples of non-LL(1) grammars that are non ambiguous, for example the one for Expressions in Chapter 6:

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

However, in this case the grammar for matched parentheses **is** *ambiguous. To convince a reader, it is necessary only to find one example of a string that can be parsed in two different ways, and show the two ways. ()()() is such a string. This would come about from a sentential form AAA, but this form AAA could have been reached in two ways:*

A = AA = AA A

or

A = AA = A AA

(g) Steven Spielberg is about to film his latest blockbuster "Elevator 9/11", in which a group of gorgeous Americans (guess what) are trapped in an elevator (a lift) hundreds of feet above the ground by a group of ugly … (you guessed right again). He wants to make sure he understands how to arrange for the elevator to get stuck. (So would you if you had the chance to spend a few hours with some of the hunks and bimbos he has lined up to act for him).

Mr Spielberg has been reliably informed that some high-rise skyscrapers have elevators that are based partway up the building, say on level (floor) X. People entering one of these elevators have a choice of pushing an UP key or a DOWN key as many times as they like. If the elevator is behaving normally it can go up or down one level for each push of the appropriate key, but it cannot drop below level X, and at the end of the day it is supposed to have returned to level X again. Write down the productions for a simple grammar that will allow Mr S to recognize a sequence of commands that does not trap the elevator, such as

UP UP DOWN DOWN or UP UP DOWN UP DOWN UP DOWN DOWN

Unusually it is sequences that should be rejected by the grammar, such as

DOWN UP or UP UP DOWN

that are the ones that really interest Mr S. (Hint: Treat UP and DOWN as key words!) [4 marks]

Oh what fun! Only a few people see through all the hot air to the underlying simplicity. The keypresses have to balance in the same way that the parentheses in the previous questions had to balance, so all it needs is the same grammar, essentially, as for matched parentheses, save that we use UP and DOWN in place of (and):

NoJam = "UP" NoJam "DOWN" NoJam | ε .

Solutions submitted on the lines of

NoJam = "UP" { NoJam } "DOWN" | "DOWN" { NoJam } "UP" .

might be suggested, but one is not allowed to drop below the initial level. Solutions like

NoJam = "UP" { "UP" "DOWN" } "DOWN" .

NoJam = { "UP" { "UP" "DOWN" } "DOWN" } .

match the two correct examples suggested, but fall into the trap of defining a grammar that only describes a subset of the language, and not the complete language.

A4. The following set of productions attempts to describe a mixed goods and passenger train:

```
Train = LocoSection [ GoodsSection ] HumanSection .
LocoSection = "loco" { "loco" } .
GoodsSection = "open" { "fuel" { "fuel" } "open" | "open" } .
HumanSection = "guard" | { "coach" } "brake" .
```

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

```
EOFSym, noSym, locoSym, fuelSym, openSym, coachSym, guardSym, brakeSym
```

develop a hand-crafted recursive descent parser for recognizing valid trains. Your parser can take drastic action if an invalid train is detected - simply produce an appropriate error message and then terminate parsing. *(You are not required to write any code to implement the getSym method.)* [20 marks]

Students had seen several examples of these sorts of parsers, as well as a few (different) grammars for trains. One is looking for something like:

```
static void accept(int wantedSym, String message) {
  if (sym.kind == wantedSym) getSym();
  else {
    IO.writeString(message); System.exit(1);
  }
}

static void abort(String message) {
  IO.writeString(message); System.exit(1);
}

static void Train() {
  LocoSection();
  if (sym.kind == openSym) GoodsSection();
  HumanSection();
}

static void LocoSection() {
  accept(locoSym, "loco expected");
  while (sym.kind == locoSym) getSym();
}

static void GoodsSection() {
  accept(openSym, "open truck expected");
  while (sym.kind == fuelSym || sym.kind == openSym) {
    switch (sym.kind) {
      case fuelSym:
        getSym();
```

```
        while (sym.kind == fuelSym) getSym();
        accept(openSym, "open truck expected");
        break;
      case openSym:
        getSym();
        break;
    }
  }
}

static void HumanSection() {
  switch (sym.kind) {
    case guardSym:
    case brakeSym:
      getSym;
      break;
    case coachSym:
      while (sym.kind == coachSym) getSym();
      accept(brakeSym, "brake coach expected");
      break;
    default:
      abort("unacceptable human part");
      break;
  }
}
```

or, in place of the last one:

```
static void HumanSection() {
  if (sym.kind == guardSym)
    getSym();
  else {
    while (sym.kind == coachSym) getSym();
    accept(brakeSym, "brake coach expected");
  }
}
```

A5. The university data processing division is faced with a problem. At short notice they have been asked

(a) to prepare a list of the surnames of the staff who have PhD degrees, and
(b) to determine the percentage of staff that have this qualification.

A data file is available with a list of staff which includes entries like

Mr Paddy O'Toole.
Ms Ima Raver, BA.
Prof Dedley Serious, BSc(Hons), MSc, PhD, PhD.
Dr Heilee-Unlykelee, PhD.
John Paul George Ringo Muggins. Prof Jo Bloggs, BSc.

Develop an attributed Cocol grammar for solving this problem. Output for the data file above should be on the lines of

The following 2 staff (33%) have PhD degrees:

Serious
Heilee-Unlykelee

Assume that

(a) names follow the patterns suggested above (start with an uppercase letter, may contain other letters or hyphens or apostrophes),
(b) titles would be limited to Mr/Miss/Ms/Mrs/Dr/Prof,
(c) degrees would be limited to BA, BSc, BA(Hons), BSc(Hons), MA, MSc or PhD.

A skeleton of a Cocol description is given below. A machine readable version of this can be found in the exam kit (`STAFFLIST.ATG`), should you prefer to hand in a machine readable solution. [24 marks]

Hints:

"Keep your grammar as simple as possible, but no simpler".

Pay particular attention to where you introduce the actions/attributes; do not simply write them in random positions in the grammar.

This is fairly simple, and I believe possible within a 24 minute time frame. In practicals and tutorials students had seen several examples of grammars of about this complexity with simple actions. The tokens representing names had been discussed as examples in class for the benefit of those who bothered to attend lectures.

```
import Library.*;

COMPILER StaffList $CN

  static ArrayList list = new ArrayList();
  static String surname;
  static boolean hasPhD;
  static int total = 0;

  CHARACTERS
    ULetter = "ABCDEFGHIJKLMNOPQRSTUVWXYZ" .
    LLetter = "abcdefghijklmnopqrstuvwxyz" .

  TOKENS
    Name = ULetter { LLetter | "'" ULetter | "-" ULetter } .

  IGNORE CHR(0) .. CHR(31)

  PRODUCTIONS
    StaffList
    = { FullName }                          (. if (total > 0) {
                                                 int n = list.size();
                                                 IO.write("The following " + n);
                                                 IO.write(" staff (" + 100 * n / total);
                                                 IO.writeLine("%) have PhD degrees:");
                                                 for (int i = 0; i < n; i++)
                                                   IO.writeLine((String)list.get(i));
                                               } .)
    .

    FullName
    = [ Title ] Names Degrees SYNC "."      (. total++; if (hasPhD) list.add(surname); .)
    .

    Title
    =   "Mr" | "Miss" | "Ms" | "Mrs"
      | "Dr" | "Prof"
    .

    Names
    = OneName { OneName } .

    Degrees                                 (. hasPhD = false; .)
    = { "," OneDegree } .

    OneDegree =   "BA" | "BSc"
                | "BA(Hons)" | "BSc(Hons)"
                | "MA" | "MSc"
                | "PhD"                     (. hasPhD = true; .)
    .

    OneName = Name                          (. surname = token.val; .)
    .

  END StaffList.
```

A6. (a) 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 in question A1 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 Java program (consider the features of Java and Parva that support the use of such a technique, and identify features that seem to act against it). [3 marks]

The insight called for here is that in Java 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).

(b) The Parva compiler supports the integer and Boolean types. Integer constants (literals) are represented in the usual "decimal" system, for example 1234.

Suppose we wished to allow hexadecimal and binary representations of integer literals as well, for example 012FH and 01101%. Which of the phases of compilation identified in question A1 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 had clearly confused "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 = Integer.parseInt(token.val, base);
                                            } catch (NumberFormatException e) {
                                              value = 0; SemError("number too large");
                                            } .)
  .
```

(c) The Cocol grammar for constructing the Parva compiler contains the productions below, which are used to handle the declaration (and possible initialization) of variables.

```
VarDeclarations<StackFrame frame>      (. int type; .)
=  Type<out type>
   OneVar<frame, type>
   { "," OneVar<frame, type> } ';' .

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++; .)
   [ AssignOp                          (. CodeGen.loadAddress(var); .)
     Expression<out expType>           (. if (!compatible(var.type, expType))
                                            SemError("incompatible types in assignment");
                                          CodeGen.assign(var.type); .)
   ]                                   (. Table.insert(var); .)
.
```

The symbol table insertion for each variable is made at the end of the `OneVar` production. A student has queried whether this production could have been changed to read as follows, where the symbol table entry is made earlier. The lecturer, by way of reply, asked her to consider what the outcome would be of trying to compile the statement

```
int i = i + 1;
```

What would your considered reaction be? [3 marks]

```
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;
                                         Table.insert(var);   // +++++ moved
                                         frame.size++; .)
   [ AssignOp                         (. CodeGen.loadAddress(var); .)
     Expression<out expType>          (. if (!compatible(var.type, expType))
                                           SemError("incompatible types in assignment");
                                         CodeGen.assign(var.type); .)
   ]
.
```

The insight looked for here escaped many candidates. If one moves the "insert" action earlier, an initialization of the form

```
int i = i + 1;
```

although problematic, will be acceptable - that is, would not generate any error messages, even though the "initialization" would be pretty meaningless. If the "insert" action is left until the end, an attempted initialization like the above would result in an "undeclared identifier" error message when the i was encountered as part of the Expression on the right of the = sign. Okay, so here is another meaningless initialization that would not be detected either:

```
int x, i = x + 1;
```

which is sad. Detecting the use of uninitialized variables is more complicated than it might at first appear!

Some candidates who got this question wrong laboured under the misconception that code like the above would store a value for i in the symbol table. Not so. However, a Parva ConstDeclaration, such as

```
const BattleOfTrafalgar = 1805;
```

would have stored 1805 in the symbol table. The two kinds of declarations are quite different.

Section B [90 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 had to happen! Over the last eight weeks a team of over 60 dynamic young programmers (DYPs) have been developing sophisticated applications in the new wonder language Parva. Like all DYPs, they have paid far too little attention to documenting what they have done. Unfortunately the day of reckoning is upon them: a directive has gone out from the Project Documentation Tyrant (PDT) that in a mere 24 hours time they will be required to provide a full set of documentation for their code, preferably in the form of a set of HTML (web) pages, one for each completed program.

Their PM (Project Manager) recently browsed through a text on Java and was struck by the existence of a utility called *JavaDoc*. This is a system program that can parse Java source code files and extract documentation from it, if such documentation is embedded in the files in a systematic way. Any "documentation" stored in a source code file must, of course, be distinguishable from the "real" source code which a compiler would process, so the system relies on extracting the documentation from comments - which, however, must not be confused with other commentary in the program. For the JavaDoc utility this is achieved by allowing commentary to take one of three forms:

```
/*  This is a typical comment in familiar form  */

//  This is a typical simple one-line comment in familiar form
```

```
/** This is a special JavaDoc comment, where the leading ** acts as
    a special marker to distinguish it from a simple comment
*/
```

Of course, JavaDoc comments would be ignored (along with other comments) if the programs were to be compiled by the standard Java compiler itself.

After consulting the PDT, the PM has asked that the team can meet the deadline by developing a utility called ParvaDoc that will do the same sort of thing for Parva. She realises that the ParvaDoc system can be generated by using Coco/R to create a "compiler" that will parse Parva programs - but rather than generate PVM code, this compiler will ignore most of the usual source code and concentrate almost entirely on certain of the comments. The PDT has agreed that the submission of an attributed Cocol grammar for the ParvaDoc utility will serve as proof that the deadline can be met.

Very generously, the PDT has supplied some examples of simple Parva programs marked up in this way, along with the form of output that he suggests should be generated - and, of course, he has also provided a simple Cocol version of the basic grammar for Parva, a copy of the Coco/R compiler generator, and a fully working executable for a Parva compiler. All this should help those 65 DYPs whose qualifications are at stake.

The crucial "ParvaDoc" comments are bound by structural rules of their own. For the task at hand, assume that they can appear only (a) immediately at the start of the code or (b) immediately at the start of a function or method. These comments are intended to document such things as the overall purpose of the application and/or its individual functions, the author, the version, the intent of the parameters or arguments, the value returned (for functions other than "void" ones), and possible references to other documents or resources.

Here is a typical example of ParvaDoc commentary for a function:

```
int ReduceLength(int[] list, int n, int length) {
/** @purpose      Reduces a list of integer numbers by removing every n-th number
    @param length specifies the original length of the list
    @return       length of reduced list
    @param n      specifies that every n-th number is to be removed
    @see          "Algorithms for a Special Sunday" (2005)
    @param list   is a reference to the list of numbers
    @version      1234
    @author       PDT
*/
  ...
}
```

As can be seen, the different components of the documentation are introduced by tags like `@purpose` and `@author`. Not all of these sections need be present, and they may be introduced in any order.

After processing the source file, the ParvaDoc system should produce output like

```
int ReduceLength (int[] list, int n, int length)

Purpose:
 Reduces a list of integer numbers by removing every n - th number

Author:
 PDT

Version:
 1234

References:
 "Algorithms for a Special Sunday" (2005)

Returns:
 length of reduced list

Parameters:

 length specifies the original length of the list

 n specifies that every n - th number is to be removed

 list is a reference to the list of numbers
```

in which we can note that the output consists of the function signature, followed by a sorted and nicely formatted list of the components forming the documentation. (A more extensive set of examples of input and output can be found in the files that have been provided by the PDT.)

The system should provide some minimal checking - for example, that tags like `@param` and `@return` are not applied to an introductory ParvaDoc comment describing an application as a whole, and that the identifiers quoted at the start of any `@param` clauses match those in the function signature. An example of the output from an attempt to analyse an incorrectly documented application might be

```
   1 /**
   2     @author  P.D. Terry, Rhodes University, 2005
   3     @return  Tomorrow's oil price
****     ↑ not applicable here
   4     @param   Humidity in Gulf of Mexico
****     ↑ not applicable here
   5 */
   6
   7 void Speculate(int[] sharePrice, int n) {
   8 /**
   9     @param   max is the gold price two weeks from today
****             ↑ max not in parameter list
  10     @return  to sanity after Wall Street crashes
****     ↑ void functions cannot return values
  11 */
  12 }
  13
  14 /** ParvaDoc comments cannot appear in arbitrary places between functions! */
**** ↑ EOF expected
```

Producing a simple text file (say `Applic.txt`) from a program (say `Applic.pav`) that contains such documentation might be a useful first step. However, as mentioned earlier, a better system would produce an HTML version (say `Applic.htm`) with HTML tags inserted to allow a browser to display the documentation in a convenient form. This is not particularly difficult to do. If you go on to derive a system to generate this format you might like to study the simple web page supplied in the files kit for the examination, which exemplifies the sorts of HTML tags that might be incorporated into the output file.

Lastly, a system like ParvaDoc is easily enhanced to allow users to insert their own HTML tags into the documentation comments. For example, if the ParvaDoc comment above were extended to have clauses reading

```
int ReduceLength(int[] list, int n, int length) {
/** ...
    @see @<a href=sunday.htm>"Algorithms for a Special Sunday" (2005)@</a>
    @author PDT @<i>with a little help from my friends@</i>@<ul>
        @<li>John
        @<li>Paul
        @<li>George
        @<li>Ringo
    @</ul>
    ...
*/
```

a hyperlink to "`sunday.htm`" would appear in the web page in the "References:" section, and the names of the four friends would be rendered by the browser as a bulleted list in the "Author:" section. It is suggested that HTML tags can be introduced into commentary in this way with an `@` character immediately preceding the tag, in the same way that the `@` character is the first one in a tag like `@see`.

Hints: Do not alter the basic Parva grammar more than is absolutely necessary - do not be tempted to add the sorts of extensions that have formed part of the practical course this year. If you cannot develop a solution that correctly deals with all three kinds of comments, try to develop one assuming that the source program will only incorporate "ParvaDoc" comments. It is not required that you produce a solution that can derive both "simple text" and/or "html" text - develop a solution for one of these only.

This is an example that is choc-a-block with opportunities for students to come up with innovative solutions, and for stronger ones to produce really sophisticated solutions. The one below is simple in concept, but relies on the candidate appreciatiing the use of the ANY keyword, which, as it happened, not many students understood, in spite of various hints being given to that effect within the case study examples given to them the day before the exam.

The essential components of the attributed grammar needed are shown below, with some explanatory comments. This is a C# version. The Java one is essentially the same, of course.

```
using Library;
using System.Text;
using System.Collections;

COMPILER TextDoc $CN

/* Parva level 2 grammar  - Coco/R for C#
   P.D. Terry, Rhodes University, 2005
   Attributed to construct ParvaDoc in flat text form */
```

We need some static variables in the parser

```
  public static OutFile output;
  static string ls = System.Environment.NewLine;
```

The following are used to help process the pragmas correctly, and the current StringBuilder is used to "collect" the tokens that form part of one component of the documentation at a time.

```
  static bool inDoc = false;
  static bool haveLS = false;
  static StringBuilder current;
```

Each of the components of a "ParvaDoc" comment is finally displayed using the following method (in passing, many candidates were guilty of horrific "cut and paste" duplication of this idea!)

```
  static void Display(string topic, StringBuilder sb) {
    if (sb.Length != 0) {
      output.WriteLine(topic);
      output.WriteLine(sb.ToString());
    }
  }
```

Tokens are added to the current StringBuilder by the following method. The trick here, which not everyone will see, is that one has to reinsert line separators which are detected as pragmas, and cannot be ignored (unless one is prepared to accept fairly messy looking documentation with potentially very long lines).

```
  static void Append(string s) {
    current.Append(s);
    if (haveLS) {
      current.Append(ls);
      haveLS = false;
    }
  }

CHARACTERS
  lf          = CHR(10) .
  backslash   = CHR(92) .
  control     = CHR(0) .. CHR(31) .
  letter      = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .
  digit       = "0123456789" .
  stringCh    = ANY - '"' - control - backslash .
  charCh      = ANY - "'" - control - backslash .
  printable   = ANY - control .
```

The next two are needed to handle the rather awkward regexp needed to define the / … */ comments as pragmas so that they can be ignored. Regexps like this were discussed in class examples, though whether all the students who skipped classes will have seen them is somewhat debatable.*

```
  nostar      = ANY - "*" .
  incomment   = ANY - "*" - "/" .

TOKENS
  identifier  = letter { letter | digit | "_" } .
  number      = digit { digit } .
  stringLit   = '"' { stringCh | backslash printable } '"' .
  charLit     = "'" ( charCh   | backslash printable ) "'" .
```

We need these two pragmas to deal with comments and line feeds. The class saw simple comments treated as pragmas in laboratory exercises.

```
PRAGMAS
  comment = "/*" nostar { incomment | "/" | "*" {"*"} incomment } "*" {"*"} "/" .

  eol  = lf .                                  (. if (inDoc) haveLS = true; .)

COMMENTS FROM "//" TO lf

IGNORE control - lf
```

Several of the productions are easily attributed. The new non-terminal DocComment is an optional new start to the grammar. The strange arguments are to allow a common DocComment production to serve in both this position and in the (more usual) one at the start of a function.

```
PRODUCTIONS
  TextDoc
  = [ DocComment<new ArrayList(), "", false>  (. output.WriteLine(); output.WriteLine(); .)
    ]
    {   FuncOrVarDeclarations
      | ConstDeclarations
    } .
```

The function declaration has to be attributed in such a way that we can build up the function header.

```
  FuncOrVarDeclarations                        (. StringBuilder header = new StringBuilder();
                                                  string type; .)
  = Type<out type>                             (. header.Append(type); .)
    Ident                                      (. header.Append(" " + token.val); .)
    (   Function<header, type>
      | GlobalVars ) .
```

If it really is a function, we need to incorporate the argument list into the function header and build up the simple symbol table of the formal argument identifiers

```
  Function<StringBuilder header, string type>
                                               (. ArrayList formals = new ArrayList(); .)
  = "("                                        (. header.Append(" ("); .)
    FormalParameters<header, formals>
    ")"                                        (. header.Append(")");
                                                  output.WriteLine();
                                                  output.WriteLine(header.ToString()); .)
```

And we have to extend the basic grammar to allow the DocComment non terminal to appear either just before or just after the opening brace of the function:

```
    (   "{" [ DocComment<formals, type, true> ]
      | DocComment<formals, type, true> "{" )
    { Statement } "}"                              .
```

The bulk of the work in this system is done within the DocComment method. We declare StringBuilders to allow us to construct each of the possible components of the documentation, and set current = purpose to allow any "free form" text at the start to be a "purpose" comment by default:

```
  DocComment<ArrayList formals, string type, bool isFunc>
                                               (. StringBuilder
                                                    version = new StringBuilder(),
                                                    author  = new StringBuilder(),
                                                    purpose = new StringBuilder(),
                                                    args    = new StringBuilder(),
                                                    refs    = new StringBuilder(),
                                                    returns = new StringBuilder();
                                                    current = purpose; .)
  = "/**"                                      (. output.WriteLine(); haveLS = false; .)
```

The body of the method is a simple loop with several options determined by tags that simply manipulate which is to be the current StringBuilder. The @param option has a simple check that the identifier is found in the little symbol table:

```
    {  "@param"                          (. if (!isFunc) SemError("not applicable here"); .)
       Ident                             (. current = args;
                                            args.Append(ls + " " + token.val);
                                            if (isFunc && !formals.Contains(token.val))
                                              SemError(token.val + " not in parameter list"); .)
     | "@version"                        (. current = version; .)
     | "@author"                         (. current = author; .)
     | "@purpose"                        (. current = purpose; .)
     | "@see"                            (. current = refs; .)
     | "@return"                         (. if (!isFunc) SemError("not applicable here");
                                            current = returns;
                                            if (type.Equals("void"))
                                              SemError("void functions cannot return values"); .)
```

Crucial to this system is the use of ANY to allow all the actual text of the documentation to be appended to the current StringBuilder!

```
     | ANY                              (. Append(" " + token.val); inDoc = true; .)
```

With a bit of care one can manipulate the spacing around some punctuation to get a better looking output, but this is a subtle point that I expect not many candidates will see:

```
     | ( "," | "." | ";" | "?" | "!" | ")" )  (. Append(token.val); .)
     | "(" ANY                          (. Append(" (" + token.val); .)

   )
```

Once the end of the DocComment is reached we can arrange to dump out the contents of the various StringBuilders, suitably labelled:

```
    "*/"                                (. Display("Purpose:", purpose);
                                           Display("Author:", author);
                                           Display("Version:", version);
                                           Display("References:", refs);
                                           if (isFunc) Display("Returns:", returns);
                                           if (isFunc) Display("Parameters:", args);
                                           inDoc = false;
                                        .) .
```

The remaining methods here are parameterized in a way that allows one to construct the function header and build up the argument list:

```
  FormalParameters<StringBuilder header, ArrayList formals>
  = [ OneParam<header, formals>
      { ","                             (. header.Append(", "); .)
        OneParam<header, formals>
      }
    ] .

  OneParam<StringBuilder header, ArrayList formals>
                                        (. string type; .)
   = Type<out type>                     (. header.Append(type); .)
     Ident                              (. header.Append(" " + token.val);
                                           formals.Add(token.val); .)
  .

  VarDeclarations                       (. string type; .)
  = Type<out type>
    OneVar { "," OneVar } ";" .

  Type<out string type>                 (. type = null; .)
  =   "void"                            (. type = "void"; .)
    | BasicType                         (. type = token.val; .)
      [ "[]"                            (. type = type + "[]"; .)
      ] .

  /* and, of course, all the other productions that remain in the grammar unaltered */

END TextDoc.
```

Here is a (Java) solution that will write web pages. It is, naturally, conceptually almost identical. The differences arise from the need to distribute a few HTML tags at judicious points into the various StringBuffers.

```
import Library.*;
import java.util.*;

COMPILER HTMLDoc $CN

/* Parva level 2 grammar  - Coco/R for Java
   P.D. Terry, Rhodes University, 2005
   Attributed to construct ParvaDoc in HTML form */

  public static OutFile output;
  static final String ls = System.getProperty("line.separator");
  static boolean inDoc = false;
  static boolean haveLS = false;
  static StringBuffer current;

  static void display(String topic, StringBuffer sb) {
    if (sb.length() != 0) {
      output.writeLine(ls + "<b>" + topic + "</b>" + ls + "<blockquote>");
      output.writeLine(sb.toString());
      output.writeLine("</blockquote>");
    }
  }

  static void append(String s) {
    current.append(s);
    if (haveLS) {
      current.append(ls);
      haveLS = false;
    }
  }

CHARACTERS
  lf          = CHR(10) .
  backslash   = CHR(92) .
  control     = CHR(0) .. CHR(31) .
  letter      = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .
  digit       = "0123456789" .
  stringCh    = ANY - '"' - control - backslash .
  charCh      = ANY - "'" - control - backslash .
  printable   = ANY - control .
  nostar      = ANY - '*' .
  incomment   = ANY - "/" - "*".
  inhtml      = ANY - ">" .

TOKENS
  identifier  = letter { letter | digit | "_" } .
  number      = digit { digit } .
  stringLit   = '"' { stringCh | backslash printable } '"' .
  charLit     = "'" ( charCh   | backslash printable ) "'" .
  html        = "@<" { inhtml } ">" .

PRAGMAS
  comment = "/*" nostar { incomment | "/" | "*" {"*"} incomment } "*" {"*"} "/" .
  eol  = lf .                                 (. if (inDoc) haveLS = true; .)

COMMENTS FROM "//" TO lf

IGNORE control - lf

PRODUCTIONS
  HTMLDoc
  = [ DocComment<new ArrayList(), "", false> ]
    {   FuncOrVarDeclarations
      | ConstDeclarations
    } .

  FuncOrVarDeclarations                       (. StringBuffer header = new StringBuffer();
                                                 String type; .)
  = Type<out type>                            (. header.append("<h3>" + type); .)
    Ident                                     (. header.append(" " + token.val); .)
    (   Function<header, type>
      | GlobalVars ) .

  Function<StringBuffer header, String type>
                                              (. ArrayList formals = new ArrayList(); .)
  = "("                                       (. header.append(" ("); .)
    FormalParameters<header, formals>
    ")"                                       (. header.append(")</h3>");
                                                 output.writeLine("<p><hr><p>");
                                                 output.writeLine(header.toString());
                                                 output.writeLine("<blockquote>"); .)
```

```
    (   "{" [ DocComment<formals, type, true> ]
      | DocComment<formals, type, true> "{" ) (. output.writeLine("</blockquote>"); .)
    { Statement } "}" .

  DocComment<ArrayList formals, String type, boolean isFunc>
                                          (. boolean par = false;
                                             StringBuffer
                                               version = new StringBuffer(),
                                               author  = new StringBuffer(),
                                               purpose = new StringBuffer(),
                                               args    = new StringBuffer(),
                                               refs    = new StringBuffer(),
                                               returns = new StringBuffer();
                                               current = purpose; .)
  = "/**"                                 (. output.writeLine(); haveLS = false; .)
    {  "@param"                           (. if (!isFunc) SemError("not applicable here"); .)
       Ident                              (. current = args;
                                             if (par) append("<p>");
                                             par = true;
                                             append(ls + "<b>" + token.val + "</b>");
                                             if (isFunc && !formals.contains(token.val))
                                               SemError(token.val + " not in parameter list"); .)
     | "@version"                         (. current = version; append(ls); .)
     | "@author"                          (. current = author; append(ls);  .)
     | "@purpose"                         (. current = purpose; append(ls); .)
     | "@see"                             (. current = refs; .)
     | "@return"                          (. if (!isFunc) SemError("not applicable here");
                                             current = returns; append(ls);
                                             if (type.equals("void"))
                                               SemError("void functions cannot return values"); .)
     | ( "," | "." | ";" | "?" | "!" | ")" )  (. append(token.val); .)
     | "(" ANY                            (. append(" (" + token.val); .)
     | ANY                                (. append(" " + token.val); inDoc = true; .)
     | html                               (. append(token.val.substring(1)); .)
    }                                     (. append(ls); .)
    "*/"                                  (. display("Purpose:", purpose);
                                             display("Author:", author);
                                             display("Version:", version);
                                             display("References:", refs);
                                             if (isFunc) display("Returns:", returns);
                                             if (isFunc) display("Parameters:", args);
                                             inDoc = false;
                                          .)
  .

  FormalParameters<StringBuffer header, ArrayList formals>
  = [ OneParam<header, formals>
      { ","                               (. header.append(", "); .)
        OneParam<header, formals>
      }
    ] .

  OneParam<StringBuffer header, ArrayList formals>
                                          (. String type; .)
   = Type<out type>                       (. header.append(type); .)
     Ident                                (. header.append(" " + token.val);
                                             formals.add(token.val); .)
  .

  VarDeclarations                         (. String type; .)
  = Type<out type>
    OneVar { "," OneVar } ";" .

  Type<out String type>                   (. type = null; .)
  =   "void"                              (. type = "void"; .)
    | BasicType                           (. type = token.val; .)
      [ "[]"                              (. type = type + "[]"; .)
      ] .

  /* and, of course, all the other productions that remain in the grammar unaltered */

END HTMLDoc.
```

In a rather different approach to solving this exercise, some candidates tried something like

```
CHARACTERS
  InDocComment = ANY - "*" .
TOKENS
  CommentString = InDocComment { InDocComment } .
PRODUCTIONS
  ...
  DocComment = "/**"
               CommentString   (. Process(token.val) .)
               "*/" .
```

where the ProcessMethod involved the use of an object of the StringTokenizer class, and a loop that extracted tokens from the CommentString. While this could be made to work (and some very nearly managed it quite well) it really involved a lot of unnecessary effort. The solution suggested earlier extracts the tokens quickly and efficiently - after all, the whole point of a tool like Coco/R is to reduce the effort of writing parsers and scanners!