RHODES UNIVERSITY

Computer Science 301 - 2011 - Programming Language Translation

Well, here you are. Here is the free information you have all been waiting for, with some extra bits of advice:

- Don't panic. It may be easier than you might at first think.
- The problems in the examination based on the exercise posed below will need rather careful thought. I shall be looking for evidence of mature solutions, not crude hacks.
- Work in smaller, rather than larger groups. Too many conflicting ideas might be less helpful than a few carefully thought out ones.
- Do make sure you get a good night's sleep!

How to spend a Special Sunday

From now until about 22h30 tonight, Computer Science 3 students have exclusive use of the Hamilton Laboratory. You are encouraged to throw out anyone else who tries to use it, but we hope that this does not need to happen. At about 22h30 Chris and I will have to rearrange things for Monday. If there is still a high demand we shall try to leave some computers for use until later, but by then we hope you will have a good idea of what is involved.

Once again, this year the format of the examination is somewhat different from years gone by, so as to counter what had happened where, essentially, one solution was prepared by quite a large group and then passed around to many others, who had probably not contributed to it in any real way. As in recent years (since 2006), the problem set below is only part of the story. At 16h30 you will receive further hints as to how this problem should be solved (by then you might have worked this out for yourselves, of course). You will be encouraged to study the problem further, and the hints in detail, because during the examination tomorrow you will be set further questions relating to the system - for example, asked to extend the solution you have worked on in certain ways. It is my hope that if you have really understood the material today, these questions will have solutions that come readily to mind, but of course you will have to answer these on your own!

My "24 hour exam" problems have all been designed so that everyone should be able to produce at least the basic solution, with scope given for top students to demonstrate their understanding of the subtler points of parsing, scanning and compiling. Each year I have been astounded at the quality of some of the solutions received, and I trust that this year will be no exception.

Please note that there will be no obligation to produce a machine-readable solution in the examination (in fact doing so is quite a risky thing, if things go wrong for you, or if you cannot type quickly). The tools will be provided before the examination so that you can experiment in detail. If you feel confident, then you are free to produce a complete working solution to Section B during the examination. If you feel more confident writing out a neat detailed solution or extensive summary of the important parts of the solution, then that is quite acceptable. Many of the best solutions over the last few years have taken that form.

During the first few hours you will be able to find the following files in the usual places, or by following links from the course web page:

- All the files as were made available for the practical course.
- The files `FREE1J.ZIP` and `FREE1C.ZIP`, which contain the Java and C# versions of the Coco/R system, and its support files, a basic grammar and skeleton files and test data for today's exercise. The kit also contains a PDF version of the Coco/R user manual (`CocoManual.pdf`).

At 16h30 new versions of the exam kit will be posted (`FREE2J.ZIP` and `FREE2C.ZIP`), and a further handout issued, with extra information, to help students who may be straying way off course.

Today things should be familiar. You could, for example, log onto the `D:` or `J:` drive, use `UltraEdit` or `NotePad++`, `LPRINT` to edit and printout files, use `CMAKE` and `CRUN` … generally have hours of fun. The C# system may work best on the `D:` drive, however.

Note that the exam setup tomorrow will have *no* connection with the outside world - no Google, FaceBook, ftp

client, telnet client, shared directories - not even a printer!

Today you may use the files and systems in any way that you wish, subject to the following restrictions: *Please observe these in the interests of everyone else in the class*.

(a) When you have finished working, **please** delete any files from the D: drive, so that others are not tempted to come and snoop around to steal ideas from you.

(b) You are permitted to discuss the problem with one another, and with anybody not on the "prohibited" list.

(c) You are also free to consult books in the library. If you cannot find a book that you are looking for, it may well be the case that there is a copy in the Department. Feel free to ask.

(d) Please do not try to write any files onto the C: directory, for example to `C:\TEMP`

(e) If you take the exam kit to a private machine you will need to have Java installed (or the .NET framework or equivalent to use the C# version).

I suggest that you *do* spend some of the next 24 hours in discussion with one another, and some of the time in actually trying out your ideas. You have plenty of time in which to prepare and test your ideas - go for it, and good luck.

If you cannot unpack the file, or have trouble getting the familiar tools to work (unlikely!), you may ask me for help. You may also ask for explanation of any points in the question that you do not understand, in the same way that you are allowed to ask before the start of an ordinary examination. You are no longer allowed to ask me questions about any other part of the course. Sorry; you had your chance earlier, and I cannot allow this without risking the chance of sneak questions and suggestions being called for.

If you cannot solve the problem completely, don't panic. It has been designed so that I can recognize that students have reached varying degrees of sophistication and understanding.

How you will spend a Merry Monday

Before the start of the formal examination the laboratory will be unavailable. During that time

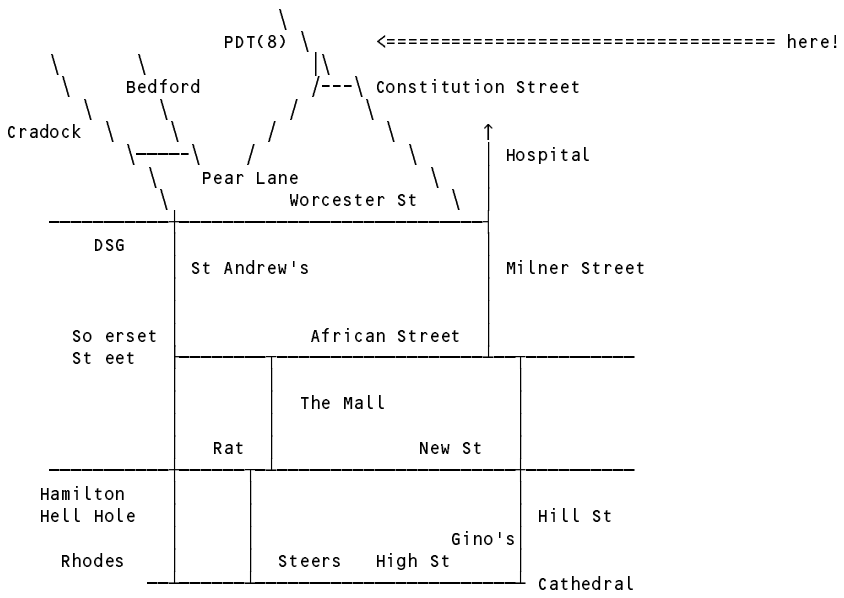
- The machines will be completely converted to a fresh exam system with no files left on directories like `D:` or `C:\TEMP` .
- The network connections will be disabled.

At the start of the examination session:

- You will receive ordinary answer books for the preparation of answers, and an examination paper.
- You will receive listings of various grammars that you received on Sunday at 8h30 and/or at 16h30. *You may annotate these during the exam to form part of your solution if you wish to submit hand-written answers to questions, and need to make reference to the code (possibly by the line numbers that are provided on the listings)*. In this case you should hand in the annotated listings with your answer book.
- You will be allocated to a computer and supplied with a `CONNECT` command for your own use. Once connected you will find an exam kit on the `J:` drive. This will contain the same Coco/R system and other files you have been given today, and in addition there will be "flat ASCII" machine readable parts of the examination paper itself, in files with names like `Q7.TXT` (Question 7). *There is no obligation to use a computer during the exam. You can answer on paper if you prefer - and yes, you can write in pencil if you prefer that*.
- At the end of the exam you will be given a chance to copy any files that you have edited or created on the D: drive back to the server. This will be explained tomorrow.
- **Remember that tomorrow you may not bring anything into the room other than your student card and writing utensils, and especially not listings, disk drives, memory sticks, text books or cell phones.**

Cessation of Hostilities Party

As previously mentioned, Sally and I would like to invite you to an informal end-of-course party at our house on Monday 14 November (after the Compilers paper). There's another copy of the wonderful ASCII-art map below to help you find your way there. It would help if you could let me know whether you are coming so that we can borrow enough glasses, plates, etc. *Please post the reply slip into the hand-in box during the course of the day.* Time: from 18h30 onwards. Dress: Casual



Preliminary to the examination

As you should know by now, having discussed this in lectures, a pretty-printer is a form of translator that takes source code and translates this into object code that is expressed in the same language as the source. This probably does not sound very useful! However, the main aim is to format the "object code" neatly and consistently, according to some simple conventions, making it far easier for humans to understand.

For example, a system that will read a set of EBNF productions, rather badly laid out as

```
Goal={One}.
One
= Two "plus" Four ".".
             Four =Five {"is" Five|(Six Seven)}.
Five = Six [Six       ].
Six=  Two| Three    |"(*" Four "*)"|'(' Four ')'.
```

and produce the much neater output

```
Goal
  = { One } .

One
  = Two "plus" Four "." .

Four
  = Five { "is" Five | ( Six Seven ) } .

Five
  = Six  [ Six ] .

Six
  = Two | Three | "(*" Four "*)" | '(' Four ')' .
```

is easily developed by using the following Cocol grammar

```
import library.*;

COMPILER EBNF $CN
/* Pretty-print a set of EBNF productions
   P.D. Terry, Rhodes University, 2011 */

  static int indentation = 0;

  public static void append(String s) {
  // Append string s to results file
    IO.write(s);
  }

  public static void newLine() {
  // Write line mark to results file but leave indentation as before
    int i = 1;
    IO.writeLine();
    while (i <= indentation) { IO.write(' '); i++; }
  }

  public static void indentNewLine() {
  // Write line mark to results file and prepare to indent further lines
  // by two spaces more than before
    indentation += 2;
    newLine();
  }

  public static void exdentNewLine() {
  // Write line mark to results file and prepare to indent further lines
  // by two spaces less
    indentation -= 2;
    newLine();
  }

  public static void indent() {
  // Increment indentation level by 2
    indentation += 2;
  }

  public static void exdent() {
  // Decrement indentation level by 2
    indentation -= 2;
  }

CHARACTERS
  letter   = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .
  lowline  = "_" .
  control  = CHR(0) .. CHR(31) .
  digit    = "0123456789" .
  noquote1 = ANY - "'" - control .
  noquote2 = ANY - '"' - control .

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

COMMENTS FROM "(*" TO "*)"  NESTED

IGNORE control

PRODUCTIONS
  EBNF
  = { Production                        (. newLine(); .)
    } EOF .

  Production                            (. String name; .)
  = NT<out name>                        (. append(name); .)
    "="                                 (. indentNewLine(); append("= "); .)
    Expression  "."                     (. append(" ."); exdentNewLine(); .)
  .

  Expression
  = Term { "|"                          (. append(" | "); .)
    Term } .

  Term
  = Factor {                            (. append(" "); .)
    Factor } .
```

```
Factor                              (. String name; .)
=   NT<out name>                    (. append(name); .)
  | terminal                        (. append(token.val); .)
  | "["                             (. append("[ "); .)
    Expression "]"                  (. append(" ]"); .)
  | "("                             (. append("( "); .)
    Expression ")"                  (. append(" )"); .)
  | "{"                             (. append("{ "); .)
    Expression "}"                  (. append(" }"); .)
.

NT<out String name>
=  nonterminal                      (. name = token.val; .)
.

END EBNF.
```

The "code generator" is embodied in the methods `append`, `indent`, `exdent` and so on, which for illustration have simply been incorporated at the start of the grammar. A more sophisticated system might direct the output to a named file in the manner familiar to you from earlier practicals, and have a `CodeGen` class.

As the first step in preparing for the examination you are invited to experiment with the above system, code for which is included in the examination kit, and then to go on to develop a pretty-printer for programs expressed in the version of Parva described in the following grammar. A version of this grammar - spread out to make the addition of actions easy - can also be found in the examination kit, along with the necessary frame files. The kit also includes a complete executable Parva compiler that matches this grammar.

```
COMPILER Parva $CN
/* Parva level 1.5 grammar for examination - Grammar only
   Java operator precedences
   Supplied Parva Compiler matches this grammar (and has a few extensions)
   P.D. Terry, Rhodes University, 2011 */

 CHARACTERS
  lf           = CHR(10) .
  backslash    = CHR(92) .
  control      = CHR(0) .. CHR(31) .
  letter       = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .
  digit        = "0123456789" .
  nonZeroDigit = "123456789" .
  stringCh     = ANY - '"' - control - backslash .
  charCh       = ANY - "'" - control - backslash .
  printable    = ANY - control .

TOKENS
  identifier   = letter { letter | digit | "_" { "_" } ( letter | digit ) } .
  number       = "0" | nonZeroDigit { digit } .
  stringLit    = '"' { stringCh  | backslash printable } '"' .
  charLit      = "'" ( charCh    | backslash printable ) "'" .

PRAGMAS
  CodeOn       = "$C+" .
  CodeOff      = "$C-" .
  DebugOn      = "$D+" .
  DebugOff     = "$D-" .
  ShortCircOn  = "$S+" .
  ShortCircOff = "$S-" .
  WarnOn       = "$W+" .
  WarnOff      = "$W-" .

COMMENTS FROM "//" TO lf
COMMENTS FROM "/*" TO "*/"

IGNORE CHR(9) .. CHR(13)

PRODUCTIONS
  Parva              = "void" Ident "(" ")" Block .
  Block              = "{" { Statement } "}" .
```

```
  Statement           = (   Block             | ConstDeclarations  | VarDeclarations | AssignmentStatement
                        | IfStatement         | WhileStatement     | ForStatement    | DoWhileStatement
                        | BreakStatement      | HaltStatement      | ReadStatement   | WriteStatement
                        | ReadLineStatement | WriteLineStatement | ";"
                      ) .
  ConstDeclarations   = "const" OneConst { "," OneConst } ";" .
  OneConst            = Ident "=" Constant .
  Constant            = IntConst | CharConst | "true" | "false" | "null" .
  VarDeclarations     = Type OneVar { "," OneVar } ";" .
  Type                = BasicType [ "[]" ] .
  BasicType           = "int" | "bool" | "char" .
  OneVar              = Ident [ "=" Expression ] .
  AssignmentStatement = Assignment ";" .
  Assignment          =   Designator ( AssignOp Expression | "++" | "--" )
                        | "++" Designator | "--" Designator .
  Designator          = Ident [ "[" Expression "]" ] .
  IfStatement         = "if" "(" Condition ")" Statement
                        { "elsif" "(" Condition ")" Statement }
                        [ "else" Statement ] .
  WhileStatement      = "while" "(" Condition ")" Statement .
  DoWhileStatement    = "do" Statement "while" "(" Condition ")" ";" .
  ForStatement        = "for" ForControl Statement .
  ForControl          =   "(" [ [ BasicType ] Ident "=" Expression ] ";" [ Condition ] ";" [ Assignment ] ")"
                        | Ident "=" Expression ( "to" | "downto" ) Expression .
  BreakStatement      = "break" ";" .
  HaltStatement       = "halt" ";" .
  ReadStatement       = "read" "(" ReadElement { "," ReadElement } ")" ";" .
  ReadLineStatement   = "readLine" "(" [ ReadElement { "," ReadElement } ] ")" ";" .
  ReadElement         = StringConst | Designator .
  WriteStatement      = "write" "(" WriteElement { "," WriteElement } ")" ";" .
  WriteLineStatement  = "writeLine" "(" [ WriteElement { "," WriteElement } ] ")" ";" .
  WriteElement        = StringConst | Expression .
  Condition           = Expression .
  Expression          = AndExp { "||" AndExp } .
  AndExp              = EqlExp { "&&" EqlExp } .
  EqlExp              = RelExp { EqlOp RelExp } .
  RelExp              = AddExp [ RelOp AddExp ] .
  AddExp              = MulExp { AddOp MulExp } .
  MulExp              = Factor { MulOp Factor } .
  Factor              = Primary | "+" Factor | "-" Factor | "!" Factor .
  Primary             =   Designator | Constant | "new" BasicType "[" Expression "]"
                        | "("
                          (   "char" ")" Factor
                            | "int" ")"  Factor
                            | Expression ")"
                          ) .
  MulOp               = "*"  | "/" | "%" .
  AddOp               = "+"  | "-" .
  EqlOp               = "==" | "!=" .
  RelOp               = "<"  | "<=" | ">" | ">=" .
  AssignOp            = "=" .
  Ident               = identifier .
  StringConst         = stringLit .
  CharConst           = charLit .
  IntConst            = number .
END Parva.
```

You should find that this is quite easy to do, leading to a system that might be able to convert a "messy" little Parva program like

```
void main(){int b,c,d;
  b=        10;
  while (b>0) { write("b = ", b); b--;
  if (b == 4) writeLine(" hit four!")
  ;       else writeLine(" on we go");}
  }
```

into the rather more readable code

```
void main() {
  int b, c, d;
  b = 10;
  while (b > 0) {
    write("b = ", b);
    b--;
    if (b == 4)
      writeLine(" hit four!");
    else
      writeLine(" on we go");
  }
}
```

Hints:

(a) Do not change the grammar to include any of the other extensions you made in Prac 25.

(b) One test for correctness is to take the output from the pretty-printer and pretty-print that again (ie take the output file from one run and use it as the input file for another run). You should, of course, just get the same result again.

(c) Another test is to compile the "raw" form of one of the demonstration programs, and also compile the "pretty" form of the same program. The `.cod` files produced by each compilation should be the same.