

RHODES UNIVERSITY

Computer Science 301 - 2017 - 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!

This year the format of the compiler examination is similar to that of the last few years. The problem set below is only part of the story. At about 16h30 you will receive further hints as to how this problem should be solved (by then I hope you might have worked this out for yourselves, of course). You will be encouraged to study the problem further, and any 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 complete working system in the examination (in fact, trying to do 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 complete working solutions 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.

How to spend a Frightening Friday

From now until about 22h30 tonight, Computer Science 3 students have exclusive use of the Hamilton Laboratories. 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 we may have to rearrange things for tomorrow. If there is still a high demand we shall try to leave some computers for use until later, but by then you should have a good idea of what is involved.

Today you will be able to find the following files in the usual places:

- All the files as were made available for the practicals, tests, solutions, past papers.
- The file FREE1.ZIP, which contains the C# versions of the Coco/R system, and its support files, some 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) and library summaries (Library.pdf).

At about 16h30 a new version of the exam kit will be posted (FREE2.ZIP), and a further handout issued, with extra information, to help students who may be straying off course.

So today things should be familiar. You could, for example, log onto the D: or J: drive, use Notepad++ and LPRINT to edit and print out files, use CMAKE to build Parva ... generally have hours of fun.

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 your files from any shared drives, 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: drive, for example to C:\TEMP\
- (e) If you take the exam kit to a private machine you will need to have the .NET framework installed.

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. If you have prepared properly for the exam you should have plenty of time in which to implement 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 Significant Splendid Saturday

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 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 a machine readable examination paper in MS Word format.
- You will receive an examination paper, with spaced questions, and you are encouraged to write your answers on the exam paper itself, or edit the MS-Word file.
- You will receive listings of various of the grammars and support files that you receive today. *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 any case you must hand back all listings with your exam paper.
- *There is no obligation to use a computer during the exam. You can answer on the examination paper if you prefer - and yes, you can write in pencil if you prefer that - but please not in red ink.*
- 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: or J: drive back to the server. This is done using a simple script and 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.**

Preliminary to the examination

I have this colleague in the Logic Department who is trying to set an exam at short notice, with questions on evaluating and simplifying Boolean expressions and building truth tables. He has suddenly realized that there are only 24 hours remaining before the students write, and that he hasn't yet solved all the problems himself, some of which have given rise to rather complicated Boolean expressions in many variables. So he is pleading for help.

My first idea was to offer him a little Cocol application that my CSC 301 students had developed very early one morning before sunrise, while they were learning about expressions and precedence and all that stuff. One of them had come up with something very promising, so I showed it to my friend. It is a Cocol description of a simple Boolean expression evaluator, with the addition of a memory of 26 cells, named by the 26 lower case letters of the standard alphabet. The evaluator would allow him to read and store values into these cells, and to compute values for expressions - which might either be printed or stored in variables. Typical input for this evaluator might read

```
a = true;
b = not a;
read(c);
read(d);
write(a or b and c or not(c == d));
```

And here is the Cocol grammar:

```
using Library;

COMPILER BoolCalc $CN
/* Simple Boolean expression evaluator with 26 memory cells - Coco/R for C#
   The nameless 63T0844, Rhodes University, 2017 */

static bool[] mem = new bool[26];

CHARACTERS
  digit    = "0123456789" .
  letter    = "abcdefghijklmnopqrstuvwxyz" .

TOKENS
  number    = digit { digit } .
  variable  = letter .

IGNORE CHR(0) .. CHR(31)

PRODUCTIONS
  BoolCalc
    = { ( Variable<out index>          (. int index = 0; bool value = false; .)
        WEAK "="
        | Expression<out value>       (. mem[index] = value; .)
        | "write" "("
        | Expression<out value>       (. IO.WriteLine(value); .)
        | { ","
        | Expression<out value>       (. IO.WriteLine(value); .)
        | } ")"
        | "read" "("
        | Variable<out index>         (. mem[index] = IO.ReadBool(); .)
        | ")"
        ) SYNC ";"
    } EOF .

  Expression<out bool value>          (. bool value1; .)
  = AndExp<out value>
    { "or" AndExp<out value1>         (. value = value || value1; .)
    } .

  AndExp<out bool value>              (. bool value1; .)
  = EqExp<out value>
    { "and" EqExp<out value1>         (. value = value && value1; .)
    } .

  EqExp<out bool value>               (. bool value1; .)
  = NotExp<out value>
    { "==" NotExp<out value1>         (. value = value == value1; .)
    | "!=" NotExp<out value1>         (. value = value != value1; .)
    } .
```

```

NotExp<out bool value>      (. value = false; .)
=   Factor<out value>
    | "not" NotExp<out value>      (. value = ! value; .) .

Factor<out bool value>      (. int index;
                             value = false;.)
=   "true"                  (. value = true; .)
    | "false"               (. value = false; .)
    | Variable<out index>    (. value = mem[index]; .)
    | "(" Expression<out value> ")" .

Variable<out int index>
= variable                  (. index = token.val[0] - 'a'; .) .

END BoolCalc.

```

"That's nice", he said. "But it's going to take a lot of typing to get all the functions into the right format, expanding where needed. It would be fantastic if I could define some functions before I start manipulating them and using them as factors in larger expressions. Like this:"

```

Fun(w, x, y, z) returns (w or x and y or not(y == z));
a = true;
b = not a;
read(c);
read(d);
write(Fun(a, b, c, d));
write(Fun(true, false, c or d, b));

```

"Well", I said, "No problem. As it happens, my students have nothing better to do in the next 24 hours, so I am sure they will all be delighted to help you!"

One of them immediately suggested that the logical way to solve the problem was simply to teach my colleague how to add Cocol code for any functions into the grammar (by extending `Factor`). For example, to incorporate an XOR function:

```

Factor<out bool value>      (. int index;
                             bool value1;
                             value = false;.)
=   "true"                  (. value = true; .)
    | "false"               (. value = false; .)
    | Variable<out index>    (. value = mem[index]; .)
*   | "XOR" "(" Expression<out value>
*   | ", " Expression<out value1> ")" (. value = value && !value1 || !value && value1; .)
    | "(" Expression<out value> ")" .

```

However, this did not appeal - my friend thinks it's illogical to have to learn Cocol and fiddle with make files and C# compilers. Like all people who leave things to the last minute he wants something very, very easy to use.

So I thought of trying a different approach: rather than obey each statement as it is analysed, and evaluate each expression as it occurs, how about a system that will generate PVM code for a simple program - which can then be executed on the PVM? That is, effectively write something like a very simple complete compiler/interpreter.

With this in mind, the first part of the exercise for today is to use Coco/R to build a system for compiling and executing a "program" like the following:

```

// User defined functions precede the statements that might require them

XOR(x, y) returns x and !y or y and !x;

AND3(x, y, z) returns x and y and z;

// definitions are followed by statements that use these functions (the "main" function in a sense)

read("Supply three values ", x, y, z);
a = AND3(x, y, z);
s = XOR(x, a);
write("and3 is ", a, " xor is ", s);

```

Showing this example to my friend in the Logic department provoked an interesting response (some people are never satisfied!)

"That's very helpful, but if those are the only statements you can use, creating truth tables is still going to be very tedious - evaluating the functions for all the combinations of w, x, y, z and so on. Can't your bright students allow me just one more feature - a simple loop structure that will allow me to cycle some variables through the values (false, true) - something like this?"

```
XOR(x, y) returns x and not y or y and not x;

// Produce a simple truth table

writeLine(" x      y      x|y      x&y      x⊕y");
loop x {
  loop y {
    writeLine(x, y, x or y, x and y, XOR(x, y));
  }
}
```

leading to output like

x	y	x y	x&y	x⊕y
false	false	false	false	false
false	true	true	false	true
true	false	true	false	true
true	true	true	true	false

"That should be easy enough, surely!" (The trouble with my colleagues is that they will keep coming back with requests for "just one more feature", but we might draw the line here for the purposes of the exam.)

In adopting this approach you can make use of the files provided in the examination kit `free1.zip` which you will find on the course website. In particular, you will find

- A grammar for Parva and support files for Parva, essentially the ones you were given at the start of Practical 7.
- Source code for the Code Generator (`CodeGen.cs`) for Parva and the Parva Virtual Machine (`PVM.cs`). It is possible to make the complete Parva compiler, if you wish to do so, in the usual way (`CMAKE Parva`).

This `CodeGen.cs` and `PVM.cs` have also been copied to the directory `LogicCom` with a change of namespace, so that you will be able to make your system with the command `CMAKE LogicCom` if your grammar is held in the file `LogicCom.atg`.

Although you are free to modify `CodeGen` and `PVM`, it is not believed that this will be necessary. You may wish to modify `LogicCom.frame`.

In addition you will find, in the root folder and the `LogicCom` folder:

- The usual frame files, and the Coco/R system;
- Some simple example programs like the one above that you can use for testing purposes;
- The evaluator grammar listed earlier (`BoolCalc.atg`).
- A skeleton grammar to act as a starting point for the present exercise, devoid of attributes (`LogicCom.atg`).
- A skeleton symbol table handler for `LogicCom` (`.\LogicCom\Table.cs`).

It is suggested that you proceed as follows:

- As with all of the "24 hour exam" problems devised over many years, this is a non-trivial exercise, and you will have to think clearly to be able to solve it.
- Remember Einstein's Advice: "Keep it as simple as you can but no simpler" and Terry's Corollary: "For every apparently complex programming problem there is an elegant solution waiting to be discovered".
- Limit the evaluator to have 26 (predeclared) variables named `a` through `z`. When they are introduced, limit function names to start with an upper case letter, so that they are easily distinguished lexically from variable

and parameter names. This makes the analysis and symbol table handling considerably easier than it might otherwise be.

- Develop the `LogicCom.atg` grammar to incorporate code generation for the PVM. Since parts of this grammar bear a resemblance to the grammar in `Parva.atg`, you may be able to do a lot of this stage by simply stealing ideas from the relevant parts of the Parva grammar (`Parva.atg`). You will need to flesh out the symbol table handler, of course. **At this first stage do not try to incorporate user-defined functions.**
- When you have done this satisfactorily you should be able to compile and execute very simple "LogicCom" programs.
- When you are happy with that, press on to develop the grammar to allow for the inclusion of function definitions and function calls. **Do not worry initially about code generation**, simply ensure that the system can parse some of the example programs in the kit (that is, don't bother to interpret them yet).
- Go on to develop the symbol table handler further, and add to the system any semantic checks needed, as well as the actions in the grammar that will generate code for the PVM for function handling.
- Your attention is drawn in particular to the opcodes `FHDR`, `CALL`, `RET` and `RETV` for calling and returning from functions, and also the `LDL` and `STL` opcodes familiar from an earlier practical.
- **Do not at first try to add other statements to the grammar** - there is enough to do simply to handle read and write and assignment statements, expressions, function definitions and function calls.
- Add the looping construct last.

The way in which the PVM handles function calls is described in chapter 14 of the text. Since you have not worked much with the new opcodes `FHDR`, `CALL` and `RET` before, it will not be giving too much away to illustrate the sort of code that should be generated for a simple function definition, and for a corresponding call of this function.

Allowing ourselves the luxury of using names and labels for illustration, rather than the numerical offsets that are required in the code proper:

`Fun(x, y, z)` returns (x or y) and z;

Fun	LDL	x	Push value of argument x
	LDL	y	Push value of argument y
	OR		Now have value of x + y at TOS
	LDL	z	Push value of argument z
	AND		Now have value of (x or y) and z at TOS
	STL	0	Store result on bottom of stack frame
	RET		Return to caller

`a = not Fun(p, q, r);`

	FHDR		Create standard stack frame
	LDL	p	Push value of first argument onto frame header
	LDL	q	Push value of second argument onto frame header
	LDL	r	Push value of third argument onto frame header
	CALL	Fun	Call function - returned value left at TOS
	NOT		Will now have the value of NOT * Fun(p, q, r) at TOS
	STL	a	Store on variable a

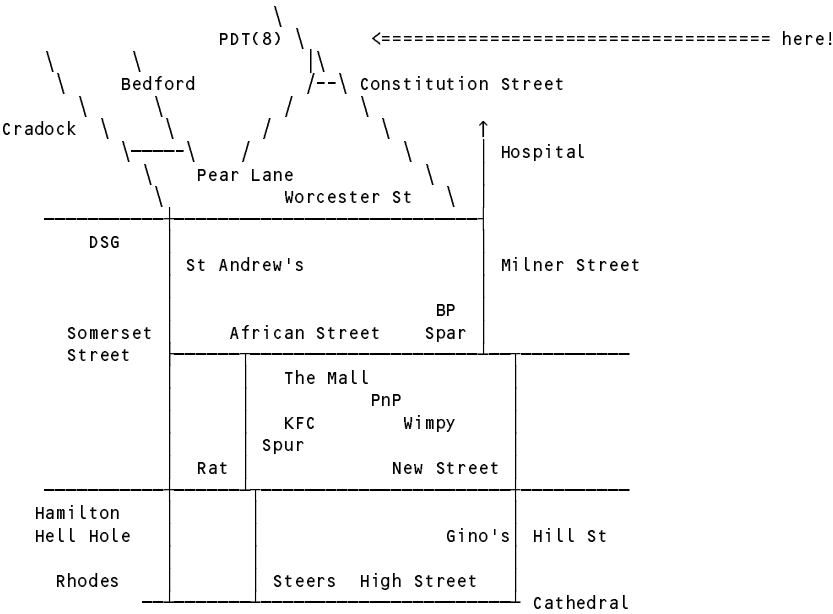
`b = Fun(true, x, y or z);`

	FHDR		Create standard stack frame
	LDC	1	Push value of first argument onto frame header
	LDL	x	Push value of second argument onto frame header
	LDL	y	
	LDL	z	
	OR		Push value of third argument onto frame header
	CALL	Fun	Call function - returned value left at TOS
	STL	b	Store on variable b

Have fun, and good luck!

Cessation of Hostilities Party

Sally and I would like to invite you to the traditional an informal end-of-course party at our house on Saturday 18 November (after the Compilers paper). There's a 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



(see also <http://www.cs.ru.ac.za/courses/CSc301/Translators/map.htm>)