

RHODES UNIVERSITY

Computer Science 301 - 2012 - 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 one 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 Chris and I may 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 I hope 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 about 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++ and 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.

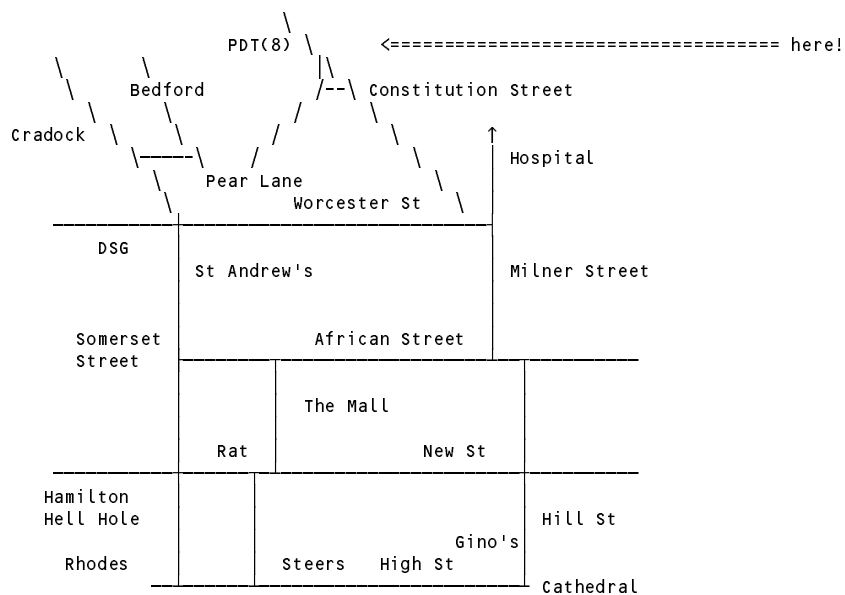
- The network connections will be disabled.

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 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 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 - but 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: 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 26 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



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

Preliminary to the examination

The code below may look vaguely familiar. It is a Cocol description of a "four function" integer calculator with the addition of a memory of 26 cells, named by the 26 lower case letters of the standard alphabet. The calculator also allows one to read values into these cells, and to compute values for expressions which might either be printed or stored in variables. Typical input for this calculator might read

```
a = 12;
b = 15;
read(c);
write(a + b * c);
```

(The C# version is almost identical, of course.)

```
import library.*;

COMPILER calc $CN
/* Simple four function calculator with 26 memory cells - Coco/R for Java
   P.D. Terry, Rhodes University, 2012 */

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

CHARACTERS
  lf      = CHR(10) .
  digit   = "0123456789" .
  letter  = "abcdefghijklmnopqrstuvwxyz" .

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

PRAGMAS
  CodeOn  = "$c+" .
  CodeOff = "$c-" .

COMMENTS FROM "//" TO lf
```

```

COMMENTS FROM "/*" TO "*/"

IGNORE CHR(0) .. CHR(31)

PRODUCTIONS

Calc
= { ( Variable<out index>          (. int index = 0; int value = 0; .)
    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.readInt(); .)
    ")"
    ) SYNC ";"
} EOF .

Expression<out int expVal>      (. int expVal1 = 0; .)
= Term<out expVal>
{ "+" Term<out expVal1>         (. expVal += expVal1; .)
  | "-" Term<out expVal1>        (. expVal -= expVal1; .)
} .

Term<out int termVal>           (. int termVal1 = 0; .)
= Factor<out termVal>
{ "*" Factor<out termVal1>       (. termVal *= termVal1; .)
  | "/" Factor<out termVal1>      (. if (termVal1 == 0) SemError("division by zero");
                                  else termVal /= termVal1; .)
  | "%" Factor<out termVal1>      (. if (termVal1 == 0) SemError("division by zero");
                                  else termVal %= termVal1; .)
} .

Factor<out int factVal>         (. factVal = 0;
                                int index, factVal2; .)
= number
{ try {
    factVal = Integer.parseInt(token.val);
  } catch (NumberFormatException e) {
    factVal = 0; SemError("number out of range");
  } .)
  | Variable<out index>          (. factVal = mem[index]; .)
  | "(" Expression<out factVal>
  ")" .

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

END Calc.

```

The exercise for the first part of today is to make some quite far-reaching extensions to this grammar and calculator to allow a user to define zero or more simple integer functions ahead of the statements that will process the expressions in terms of the variables and constants they incorporate.

In particular cases one might be tempted to do this for a known set of functions by hard-coding them into the system - for example by extending `Factor` as follows to incorporate a very simple (restrictive) `SQR` function:

```

Factor<out int factVal>         (. factVal = 0;
                                int index, factVal2; .)
= number
{ try {
    factVal = Integer.parseInt(token.val);
  } catch (NumberFormatException e) {
    factVal = 0; SemError("number out of range");
  } .)
  | Variable<out index>          (. factVal = mem[index]; .)
  | "SQR" "(" Variable<out index> (. factVal = mem[index] * mem[index]; .)
  ")"
  | "(" Expression<out factVal>
  ")" .

```

However, this does not easily generalize, and a different approach might be as follows: Change the actions in the above grammar so that, rather than obey each statement as it is analysed, the system will generate PVM code for a simple program which can then be executed on the PVM - that is, effectively write something like a subset of the extant Parva compiler, (but very much simpler).

With this in mind, the exercise for today is to use Coco/R to build a system for running a "program" like the following:

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

SQR(x) returns x * x;

Sum(x, y, z) returns x + y + z;

// Typical statements that use these functions

read("Supply three numbers ", x, y, z);
a = Sum(x, y, z);
s = SQR(x);
write("Sum is ", a, " x2 is ", s);
```

That should keep you busy for a few hours!

In adopting this approach you can make use of the files provided in the examination kit `free1j.zip` (Java version) or `free1c.zip` (C# version) which you will find on the course website. In particular, you will find

- A grammar for Parva level 1, essentially as described in the text book, chapter 13, with a few cosmetic differences and additions, and essentially the one you were given in Practical 25.
- The Symbol Table handler (Table) for Parva level 1 - essentially the same as that in the textbook, with a few cosmetic differences and additions, and essentially as you were given in Practical 25.
- A Code Generator (CodeGen) for Parva, level 2, which incorporates the extra opcodes described in Chapter 14 for calling and returning from functions - again, essentially as you were given in Practical 25
- A Parva Virtual Machine (PVM) for level 2, which can interpret the extra opcodes described in Chapter 14 for calling and returning from functions, and also the LDL and STL opcodes familiar from an earlier practical (but not the specialised ones like `LDL_3` and `LDA_0`).
- The CodeGen and PVM classes are compatible with Parva, level 1, of course, even though they incorporate the extra opcodes from Chapter 14 (which were in the kit for Practical 25 as well). Although you are free to modify them, it is not believed that this will be necessary. It is possible to make the complete Parva compiler, if you wish to do so, in the usual way (`CMAKE Parva`).

In addition you will find, in the root folder and the `CalcPVM` 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 Calculator grammar listed earlier (`Calc.atg`).
- A skeleton grammar to act as a starting point for the present exercise, devoid of attributes (`CalcPVM.atg`).
- A skeleton symbol table handler for `CalcPVM` (`.\CalcPVM\Table.java` or `.\CalcPVM\Table.cs`).

It is suggested that you proceed as follows:

- Limit the calculator to have 26 (predeclared) variables named a through z. When they are introduced, note that function names start with an upper case letter, so 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 `CalcPVM.atg` grammar to incorporate code generation for the PVM. Since parts of this grammar match the grammar in `Parva.atg`, you should be able to do quite a lot of this by simply copying

(cut/paste) 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 "four function" 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.
- **Do not 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. Although the Parva grammar given to you incorporates Boolean operators and expressions, confine yourself to user-defined functions that take integer parameters and arguments and produce integer results (but don't bother to delete the parts of the Parva grammar that handle Boolean operators).

The way in which the PVM handles function calls is described in chapter 14 of the text. Since you have not worked 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 + y) / z;

Sum	LDL	x	Push value of argument x
	LDL	y	Push value of argument y
	ADD		Now have value of x + y at TOS
	LDL	z	Push value of argument z
	DIV		Now have value of (x + y) / z at TOS
	STL	0	Store result on bottom of stack frame
	RET		Return to caller

a = 2 * Fun(x, y, z);

	LDC	2	Push 2 on stack in anticipation of later multiplication
	FHDR		Create standard stack frame
	LDL	x	Push value of first argument onto frame header
	LDL	y	Push value of second argument onto frame header
	LDL	z	Push value of third argument onto frame header
	CALL	Fun	Call function - returned value left at TOS
	MUL		Will now have the value of 2 * Fun(x, y, z) at TOS
	STL	a	Store on variable a

b = Fun(100, 50, z) - 4;

	FHDR		Create standard stack frame
	LDC	100	Push value of first argument onto frame header
	LDC	50	Push value of second argument onto frame header
	LDL	z	Push value of third argument onto frame header
	CALL	Fun	Call function - returned value left at TOS
	LDC	4	Push 4 on stack ready to subtract
	SUB		Will now have the value of Fun(100, 50, z) - 4 at TOS
	STL	b	Store on variable b

Have fun, and good luck!