RHODES UNIVERSITY

Computer Science 301 - 2014 - 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 Significant 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 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 we hope you will have a good idea of what is involved.

Once again, this year the format of the examination is somewhat 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 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, 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 of library summaries (`Library.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.

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 Java installed, or have 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. If you are well prepared for the exam you should 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 Mind-blowing Monday Morning

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 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: drives 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 Section B of the examination

As by now you are well aware, the Parva language and its compiler, as found in the textbook, were designed to serve two roles - firstly, to provide a system in which small, but non-trivial, programs could be written and, secondly, and more importantly, to act as a system which could be extended in numerous ways to allow students to learn some of the techniques of language design and implementation.

The only data types found in the basic version of Parva are integers, booleans, and one-dimensional arrays of elements of those types. During the course you have seen how the character data type can be added to this quite easily, as well as a few new statement forms.

An array structure is (as you should be aware) essentially a list of elements all of the same type, that is, a list of *homogeneous* elements. Arrays are implemented in the PVM by allowing a programmer to allocate storage for them on the "heap" as needed, and an array is accessed through a "pointer", a static variable which contains the address of the base of the array structure itself. The pointer itself, is, of course, stored on the "stack" of the PVM. If this is still a mystery to you, it is suggested that you make sure that it is no longer a mystery by the start of the formal examination.

In many applications one requires structures whose elements are *inhomogeneous*. Although this form of structure was not explicitly provided in the earliest languages like FORTRAN, BASIC and Algol, it became popular in the 1970s when Pascal introduced the concept of a *record* type. At first, such structures could contain only data elements, as exemplified by a type definition, or a model of an object with five *fields*.

```
TYPE
  Person = RECORD
             age       : INTEGER;
             salary    : REAL;
             isBearded : BOOLEAN;
             firstName, surname : ARRAY [0 .. 100] OF CHAR;
           END; (* Person *)
```

which you might find more familiar if it were written equivalently using a C#-like notation:

```
class Person {
  int    age;
  double salary;       // I wish!
  bool   isBearded;
  string firstName, surname;
} // Person
```

In C# the concept of a *class* goes much further than this; classes are permitted to package together methods as well as data and, as such, support object-oriented programming in a manner that Pascal and Modula-2 could not.

If you limit yourselves to using a *class* merely as a definition of an aggregate of data values, you can have anything up to 24 hours of instructive fun by extending the Parva system to compile and execute programs such as the following (where the labelled points refer to the notes on the next page):

```
void main() {  // supplied in the kiit as    examples\r15.pav
// Display use of a simple class and object
  const passMark  = 50;
  const firstMark = 75;

  class TestResult {                                    // point (a)
    int mark;                                           // point (b)
    bool passed, first;                                 // point (c)
  };

  int firsts = 0;

  TestResult t;                                         // point (d)
  t = new TestResult();                                 // point (e)

  while (true) {
    read("Supply mark (negative exits) ", t.mark);      // point (g)
    if (t.mark < 0) break;                              // point (f)
    t.passed = t.mark >= passMark;
    t.first = t.mark >= firstMark;                      // point (g)
    if (t.passed) writeLine("Passed comfortably");
    if (t.first) firsts++;                              // point (h, i)
  }
  writeLine(firsts, " distinctions");
} // main
```

This program illustrates various aspects of the extensions you will need to introduce, as indicated by the labelled points in the code on the previous page.

(a) The compiler will have to allow for the definition of a new kind of identifier, namely *class*. The definition may be regarded as a new kind of declaration statement, and effectively defines a new type.

(b) Within the braces associated with a *class* definition is to be found a *field list*.

(c) Items in a field list consist of a type specifier (exemplified by `bool`) and a list of fields of that type (`passed` and `first`). Assume that all such fields are "public variables", in the C#/Java sense.

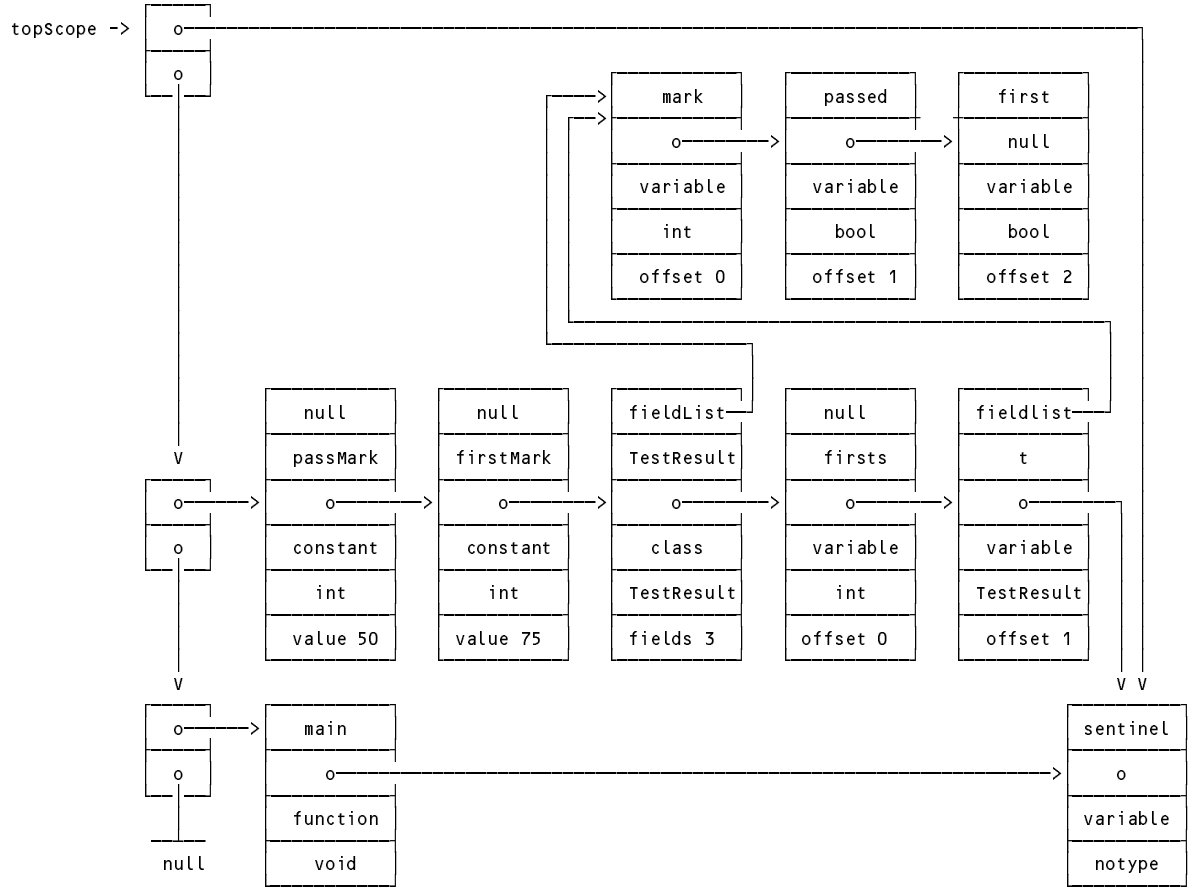
(d) The compiler will have to allow for the declaration of object variables (such as `t`) of a class type defined earlier in the code (the usual "define before use" restriction).

(e) Objects of a class type (like `t`) are to be allocated on the *heap*, just as arrays are, using a variation on the use of the keyword `new`. Limit yourself to the simple `new ClassName()` constructor, as shown above.

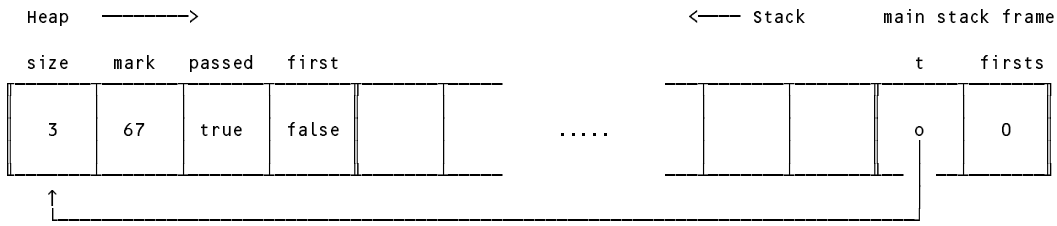
(f) It gives little away to point out that the concept of a *designator* has to be extended to allow a field of an object (as in `t.mark`) to be addressed where its value is needed in an expression like `(t.mark < 0)`.

(g) Similarly, fields of an object can be addressed so as to act as target designators within read statements (`read(t.mark)`) and assignments (`t.first = ...`).

(h) Clearly, much will depend on successfully modifying the symbol table and its ancillaries to handle all these extensions. It might help to show part of what a symbol table might look like after the *compiler* had reached the point (h) in the above code (immediately before closing off compilation of the *while* statement).



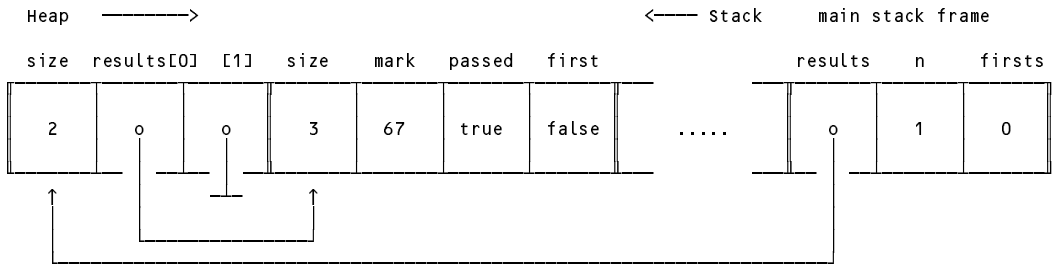
(i) For the program above it might be helpful to show the state of the stack and heap when point (h) is reached for the first time during *execution*, after a value of 67 has been read as the first mark.



Enough already?! No! The system should also allow the creation of arrays of objects, as exemplified by:

```
void main() {
// Demonstrate use of arrays of objects
  const passMark  = 50;
  const firstMark = 75;

  class TestResult {
    int mark;
    bool passed, first;
  };

  int firsts = 0;
  int n = 0;

  TestResult[] results = new TestResult[2];  // create an array of objects on the heap

  while (true) {
    TestResult t = new TestResult();
    read("Supply next mark (negative exits) ", t.mark);
    if (t.mark < 0) break;
    t.passed = t.mark >= passMark;
    t.first = t.mark >= firstMark;
    if (t.passed) writeLine("Passed comfortably");
    if (t.first) firsts++;
    results[n] = t;
    n++;                                    // point (h)
  }
  writeLine(firsts, " distinctions");
} // main
```

In this case, were the program to be *executed*, then, immediately after the `n++;` assignment at point (h) had been effected for the first time, the state of the stack and heap would be as follows:



You might like to consider what the symbol table would have looked like at the same point (h) while the program was being *compiled*.

Where do you go from here?

Take a little time to read through the rest of this document carefully and make sure you understand it before you leap in and start to hack at a "solution".

In adopting the approach suggested below you can (and should) 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

* The usual frame files, and the Coco/R system;

* A directory folder `examples` containing some example programs, like the ones given above. You may use these for testing purposes (several of them are very simple; see the attached listing);

* The files to build a Parva level 1 compiler using the Java/C# expression precedence hierarchy, that is, all the files needed to build a Parva compiler essentially identical to the one supplied to you during the final practical, with a few of the refinements made during that exercise.

Firstly, develop the Cocol grammar (`Parva.atg`) to the point where it can parse the *definitio*n of a class, that is, add the necessary syntax, without yet adding semantic actions.

Develop the grammar further so that the Parva compiler can parse the *declaration* of objects of a known class type - once again concentrating on getting the syntax specified correctly and eliminating LL(1) conflicts.

Develop the syntax of the grammar further so that the Parva compiler can parse designators, including those that incorporate an object and its fields.

Develop the syntax still further so that expressions can incorporate the `new()` operator as it applies to objects as well as the `new XX []` operator as it applies to arrays. Hint: Focus on the `Primary` non-terminal of the grammar and take care to eliminate LL(1) conflicts.

You should be able to carry out the preceding steps without paying much, if any, attention to symbol table construction, semantic checking and code generation. The next crucial step is to consider how to modify the `Kinds, Types` and `Entry` classes in the file `Parva\Table.cs` (or `Parva\Table.java`) to handle the introduction of classes and objects. Allow more than one class to be defined in a program, of course, as as you can see in some of the example programs.

Modify the `Table` class to handle any new features required (including the method for displaying the contents of the symbol table if the useful debugging pragma $ST is encountered during compilation).

Extend the parser so that it not only recognizes class definitions and object declarations, it also makes the appropriate symbol table entries for these and, as it does so, determines the size of the main method's stack frame. Compile a few programs and check that the symbol tables are correct (no need to execute them yet).

Modify the compiler so that the `Primary` parser can generate appropriate code for allocating an object of a *class* type on the heap at run-time. You will not need any new opcodes in the PVM - simply make use of the existing `PVM.anew` opcode. Compile a few programs and check that the allocation works correctly (make use of the $SD and $HD directives in debug mode, and the tracing feature in the interpreter).

Modify the compiler so that the `Designator` parser generates correct code for computing the address of an object field at run-time. Once again, you will not need any new opcodes in the PVM - simply make use of the existing `PVM.lda, PVM.ldxa` and `PVM.ldv` opcodes.

When you have done all this satisfactorily you should be able to compile and run simple complete correct programs (and also reject incorrect ones). You will also have a Parva compiler unlike any developed before!

STOP PRESS: To allow you to test your compiler on programs that are not meant to be executed (merely compiled so as to check the `.COD` file, symbol table or the program listing), the Parva compilers now recognize a `-n` command line flag, as in

```
Parva  voter.pav  -l  -d  -n
```

which won't call on the PVM to interpret the code after it has been generated. Try it!

CAUTION: 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. By way of assistance, at 12h30 I shall release a further short handout that may contain some useful hints or raise points that you might not have realized. Later still, at about 16h30 there will be a further handout. You will need to study all of this material carefully, as the formal examination tomorrow will require you to demonstrate understanding of what you do today. Good luck!