PROGRAMMING LANGUAGE TRANSLATION

FREE INFORMATION PART 1

November 2019

Welcome to the notorious 24-hour exam. 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 final solutions in Section B tomorrow will need rather careful thought. I am looking for evidence of mature solutions and insight, 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.

Extending Parva to allow for enumeration types

Most computer languages provide simple, familiar, notations for handling arithmetic, character and Boolean types of data. Variables, structures and arrays can be declared of these basic types. Some languages, notably Pascal, Modula-2, C, C++, and Ada, allow programmers the flexibility to define what are often known as enumeration types, or simply enumerations. Here are some examples to illustrate this idea:

```
TYPE  (* Pascal or Modula-2 *)
  COLOURS = ( Red, Orange, Yellow, Green, Blue, Indigo, Violet );
  INSTRUMENTS = ( Drum, Bass, Guitar, Trumpet, Trombone, Saxophone);
VAR
  Walls, Ceiling, Roof : COLOURS;
  JazzBand : ARRAY [0 .. 40] OF INSTRUMENTS;
```

with the equivalent in C or C++:

```
typedef  /* C or C++ */
  enum { Red, Orange, Yellow, Green, Blue, Indigo, Violet } COLOURS;
typedef
  enum { Drum, Bass, Guitar, Trumpet, Trombone, Saxophone} INSTRUMENTS;
COLOURS Walls, Ceiling, Roof;
INSTRUMENTS JazzBand[41];
```

Sometimes the variables are declared directly in terms of the enumerations:

```
VAR  (* Pascal or Modula-2 *)
  CarHireFleet : ARRAY [1 .. 100] OF ( Golf, Tazz, Sierra, BMW316 );
```

or

```
enum CARS { Golf, Tazz, Sierra, BMW316 } CarHireFleet[101];  /* C or C++ */
```

Java got into the act rather later; the original version of Java did not provide the facility that is now manifest in Java and C#, where we might declare:

```
enum COLOURS { Red, Orange, Yellow, Green, Blue, Indigo, Violet };
enum INSTRUMENTS { Drum, Bass, Guitar, Trumpet, Trombone, Saxophone };
COLOURS      Walls, Ceiling, Roof;
INSTRUMENTS[] JazzBand = new INSTRUMENTS[41];
```

The big idea here is to introduce a distinct, usually rather small, set of values that a variable can legitimately be assigned. Internally these values are represented by small integers - in the case of the CarHireFleet example, the "value" of Golf would be 0, the value of Tazz would be 1, the value of Sierra would be 2, and so on.

In the C/C++ development of this idea, the enumeration, in fact, results in nothing more than the creation of an implicit list of const int declarations. Thus the code:

```
enum CARS { Golf, Tazz, Sierra, BMW316 } CarHireFleet[101];
```

is semantically completely equivalent to

```
const int Golf = 0; const int Tazz = 1;
const int Sierra = 2, const int BMW316 = 3;
int CarHireFleet[101];
```

and to all intents and purposes this gains very little, other than possible readability - an assignment like

```
CarHireFleet[N] = Tazz;
```

might, of course, convey more to a reader than the semantically identical

```
CarHireFleet[N] = 1;
```

In the much more rigorous Pascal and Modula-2 approach one would not be allowed this freedom; one would be forced to write

```
CarHireFleet[N] := Tazz;
```

Furthermore, whereas in C/C++ one could write code with rather dubious meaning like

```
CarHireFleet[4] = 45; /* Even though 45 does not correspond to any car! */
CarHireFleet[1] = Tazz / Sierra;  /* Really! */
Walls = Sierra;    /* Pretty much anything is allowed in C++ */
```

in Pascal, Modula-2, Java and C# one cannot perform arithmetic on variables of these types directly, or assign values of one type to variables of an explicitly different type. In short, the idea is to promote "safe" programming - if variables can meaningfully only assume one of a small set of values, the compiler (and/or run-time system) should prevent the programmer from writing (or executing) meaningless statements.

Clearly there are some operations that could have sensible meaning. Looping and comparison statements like

```
if (Walls == Indigo) Redecorate(Blue);
```

or

```
for (Roof = Red; Roof <= Violet; Roof++) DiscussWithNeighbours(Roof);
```

or

```
if (JazzBand[N] >= Saxophone) Shoot(JazzBand[N]);
```

might reasonably be thought to make perfect sense - and would be easy to "implement" in terms of the underlying integer values.

In fact, the idea of a limited enumeration is already embodied in the standard character (and, in some languages, Boolean) types - type Boolean is, in a sense the enumeration of the values {0, 1} identified as {false, true}, although this type is so common that the programmer is not required to declare the type explicitly. Similarly, the character type is really an enumeration of a sequence of (typically) ASCII codes, and so on.

Although languages that support enumeration types forbid programmers from abusing variables and constants of any enumeration types that they might declare, the idea of "casting" allows programmers to bypass the security where necessary. The reader will be familiar with the (rather strange) notation in the C family of languages that allows code like

```
char uc = (char) 65;                  // 'A'
char lc = (char) ( (int) uc + 32 );   // 'a'
```

In Pascal and Modula-2 a standard function `ORD(x)` can be applied to a value of an enumeration type to do little more than cheat the compiler into extracting the underlying integral value. This, and the inverse operation of cheating the compiler into thinking that it is dealing with a user-defined value when you want to map it from an integer, are exemplified by Modula-2 code like

```
IF (ORD(Bagpipe) > 4) THEN .....
Roof := VAL(COLOURS, I + 5));
```

Rather annoyingly, in Pascal and Modula-2 one cannot read and write values of enumeration types directly - one has to use these casting functions and switching statements to achieve the desired effects.

Enumerations are a "luxury" - clearly they are not really needed, as all they provide is a slightly safer way of programming with small integers. Not surprisingly, therefore, they are not found in all languages.

In recent times you have studied and extended a compiler for a small language, Parva, in the implementation of which we have repeatedly stressed the ideas and merits of safe programming.

How would you add the ability to define enumeration types in Parva programs and to implement these types, at the same time providing safeguards to ensure that they could not be abused? Initially, strive to develop a system that will allow:

- the declaration of an enumeration type name and its associated values
- the declaration of variables (and arrays) of those types
- assignment of assignment-compatible values of those types to these variables
- comparison of values of compatible types for equality and ordering
- casting between types using a notation like
  ```
  Roof = cast(COLOURS, someIntegerValue);
  // i.e. interpret someIntegerValue as a member of
  // the COLOURS enumeration and assign to Roof
  ```
- incrementing and decrementing variables using the familiar ++ and -- notation
- for loops that can be controlled by variables of an enumerated type as well as by integers and characters
- values of enumerated types to be written (for simplicity, as the value of the underlying integer).

You may wish to read up a little more on enumeration types as they are used in languages like Modula-2 and Pascal. Enumeration types in Java and C# are rather more complex in their full ramifications, however.

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 `free1.zip`, which you will find on the course website. In particular, you will find, after unzipping this, a structure like that below - and it is recommended that you retain this directory structure

```
CMake.bat               Script to generate and compile the Parva compiler

CocoManual.pdf          Documentation

Coco.exe                Coco compiler and basic frame files
Scanner.frame
Parser.frame
Driver.frame
Parva.atg               Parva grammar
Parva.frame             Parva frame file

Library.cs              Rhodes C# library

Parva\CodeGen.cs        Auxiliary routines for Parva compiler
Parva\PVM.cs
Parva\Table.cs

examples\t*.pav         Source code for “enumeration” type test programs
examples\<various>.pav  Source code for basic function test programs
```

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, syntax or the program listing), remember that the Parva compiler recognizes 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. It also recognizes a `-g` command line flag, as in

```
parva  t03.pav  -l  -d  -g
```

which will follow a successful compilation with an immediate interpretation without the usual annoying questions.

The most complicated test program in the kit is given below, but there are several simpler ones that you could use to develop the system in an incremental fashion, with names like `t01.pav`, `t02.pav`, etc.

```
void main () { // enumtest.pav
// Illustrate some simple enumeration types in extended Parva
// Some valid declarations

  enum DAYS    { Mon, Tues, Wed, Thurs, Fri, Sat, Sun };
  enum WORKERS { BlueCollar, WhiteCollar, Manager, Boss };
```

```
    enum DEGREE  { BSc, BA, BCom, MSc, PhD };
    enum FRUIT   { Orange, Pear, Banana, Grape };

    const pay = 100;

    DAYS yesterday, today;
    WORKERS[] staff = new WORKERS[12];
    int[] payPacket;
    int i;
    bool rich;
    FRUIT juice = Orange;
    DEGREE popular = BSc;

  // Some potentially sensible statements
    today = Tues;
    yesterday = Mon;        // That follows!
    if (today < yesterday) write("Compiler error");   // Should not occur
    today++;               // Working past midnight?
    if (today != Wed) write("another compiler error");

    int totalPay = 0;
    for today = Mon to Fri
      totalPay = totalPay + pay;
    for today = Sat to Sun
      totalPay = totalPay + 2 * pay;

    rich = staff[i] > Manager;
    yesterday = cast(DAYS, (int) today - 1);
    DAYS tomorrow = cast(DAYS, (int) today + 1);

  // Some possible meaningless statements - be careful
    enum COLOURS { Red, Orange, Green };  // Is this valid?
    juice = cast(FRUIT, Pear);            // Is this valid?
    juice = cast(FRUIT, popular);         // Is this valid?
    Sun++;                                // Cannot increment a constant
    today = Sun; yesterday = today - 1;   // Sounds reasonable? Or is it?
    if (today == 4)                 // Invalid comparison - incompatibility
      staff[1] = rich;              // Invalid assignment - incompatibility
    Manager = Boss;                 // Cannot assign to a constant
  }
```

Hints:

(a) The examination kit includes all the files needed to build a working Parva compiler similar to the one used in your last two practical exercises (it does not have all the extensions you were asked to make in that practical, and you need not bother to try to add all of those extensions again). It does have a changed for-loop that resembles the one in Pascal, rather than the Python for statement that you implemented. It gives little away to advise you to think carefully about the form the symbol table and its entries will take.

(b) It also includes various simple test programs of the sort outlined above (in files named `tXX.pav`). The example Parva programs in the source kit range from very short, simple ones, to more advanced ones. You are well advised to start with the very simple ones that are only a few statements long!

(c) The systems allow you to use a command-line parameter -D to display the symbol table at the close of every block, and a parameter -C to request a listing of the generated code, which you might find useful.

(d) Later in the day - at 18h00 - more information will be released, to help those of you who may not have completed the exercise to do so. Section B of the examination tomorrow will include a set of unseen questions probing your understanding of the system.

(e) Rest assured that you will not be expected to reproduce a complete Parva compiler from memory under examination conditions, but you may be asked to make some additions or improvements to the system developed today. You will not be asked to make the exact extensions of the last practical again.

(f) Remember Einstein's Advice: "Keep it as simple as you can but no simpler" and the Corollary coined by Prof. Terry, the developer and instructor (for many years) of this course: "For every apparently complex programming problem there is an elegant solution waiting to be discovered".

Summary of useful library classes

See the file “Library_class_information.txt” provided in the free kit for information on useful input/output, string and list classes.