The *Circus-DTE* tutorial

A document for introducing the Circus-DTE programming language.

Written by: **David Ramsey**
Dave.Ramsey@xrce.xerox.com
Date: **Aug 2d, 2002**
Xerox Research Centre Europe
Version: **1.1**

Version Control Details

Version	Date	Author	Description of Change
0.8	2/8/99	DR	Initial draft version
0.81	4/8/99	DR	Incorporation of LH Feedback incl. Index
0.82	6/8/99	DR	Added Keywords list, operator lists
0.83	13/9/99	DR	Incorporated GH Feedback
0.9	20/9/99	DR	Brought tutorial up to date with Circus-DTE 0.2 Incorporated JY-VD Feedback
0.91	28/9/99	DR	Incorporated 1st half of Peer Review feedback
0.92	11/10/99	DR	Added examples, rephrased many paragraphs.
0.93	18/10/99	DR	Incorporated v.03 of Circus-DTE.
0.94	25/10/99	DR	2nd half of PR (6/9 completed) Fixed examples (EP Feedback). Incorporated PR feedback.
0.95	28/10/99	DR	Added Concurrency Section
0.95b	28/10/99	JYVD	Typos and grammatical corrections
1.0	20/03/02	VL	Update to reflect changes in Circus-DTE V.2
1.1	2/8/02	VL	Update to reflect changes and additions in Circus-DTE V.2

Table of Contents

1. INTRODUCTION TO THE TUTORIAL 7

1.1 FOREWORD 7
1.2 WHAT IS CIRCUS-DTE? 7
1.3 PURPOSE OF THIS DOCUMENT 7
1.4 TARGET AUDIENCE 8
1.5 REQUIREMENTS 8
1.6 CIRCUS-DTE VERSION 8
1.7 CONVENTIONS 8

2. BEGINNING CIRCUS-DTE: HELLO WORLD! 10

2.1 STEP 1: CREATING AND EDITING THE CODE 10
2.2 STEP 2. SAVING YOUR FILE 10
2.3 STEP 3. COMPILING AND RUNNING THE CODE 10
2.4 STEP 4. READING THE OUTPUT 10
2.5 WALKTHROUGH 10
2.6 TROUBLESHOOTING 11
2.7 EXPLANATIONS 11
2.8 NOTES ON THE COMPILATION 12
2.8.1 Verbose output 12
2.8.2 Output with Error Reporting 12
2.8.3 Options help for the compiler 12
2.9 SUMMARY 13

3. CIRCUS-DTE TYPES 14

3.1 WHEN TO USE THE VERIFY COMMAND 14
3.1.1 Testing with the verify command 15
3.2 BASE TYPES 15
3.2.1 The Boolean type 15
3.2.2 The Int and Float types 16
3.2.3 The String type 16
3.2.4 Bytes Type 18
3.2.5 The Multiset type 18
3.2.6 The Sequence type 19
3.2.7 The Tuple type 21
3.2.8 The Structure (Record))type 22
3.2.9 The Dictionary type 22
3.2.10 The Range Function 23
3.2.11 The None type 23
3.2.12 The Void type 24
3.2.13 The Unit Type 24
3.2.14 The Reference Type 24
3.3 POLYMORPHISM THROUGH SUB TYPES 25
3.4 TYPE DECLARATION 26
3.5 EXERCISES 27
3.6 SUMMARY 27

4. CIRCUS-DTE VARIABLES: 29

4.1.1 The differences between consts and vars 29
4.1.2 Scope of variables 30
4.1.3 Basic Built in functions 30

4.1.4 The mod function ... *31*
4.1.5 Randomising Functions ... *32*
4.1.6 Using the Built in functions ... *32*
4.1.7 Casting Variables ... *33*
4.1.8 Default Values: ... *33*
4.2 EXERCISES ... 34
4.3 SUMMARY ... 34

5. LAMBDA FUNCTIONS ... 35

5.1 DEFINING A LAMBDA FUNCTION ... 35
5.2 USING LAMBDA FUNCTIONS ... 35
5.3 CALLING A LAMBDA FUNCTION ... 35
5.4 LAMBDA PROGRAMMING ... 36
5.5 FURTHER EXAMPLES ... 36
5.5.1 A 'HelloWorld' lambda function ... *36*
5.5.2 Additional Parameters ... *36*
5.5.3 Anonymous functions ... *37*
5.6 EXERCISES ... 38
5.7 SUMMARY ... 38

6. EXPANDING THE COMPUTATIONAL STATEMENTS ... 39

6.1 THE FOR .. DO LOOP ... 39
6.2 IF, THEN, ELSE ... 40
6.3 THE' [|…|]' OPERATOR (OR ACTION SYSTEM) ... 40
6.4 THE 'MAP' OPERATOR ... 41
6.5 SUMMARY ... 41

7. POLYMORPHIC ABSTRACT MACHINES: (PAMS) ... 42

7.1 AN EXAMPLE PAM ... 42
7.2 DEFINITION ... 42
7.3 USAGE ... 42
7.4 WHY 'POLYMORPHIC' ABSTRACT MACHINES? ... 43
7.5 PARAMETER PASSING, BY COPY ... 43
7.6 COMPOSING PAMS ... 43
7.7 SUMMARY ... 43

8. PATTERN-MATCHING ... 45

8.1 THE SUBJECT ... 45
8.1.1 Matching with a Filter ... *45*
8.1.2 The '%' Filter ... *45*
8.1.3 The '?' Filter ... *46*
8.1.4 Assignment with a Filter ... *46*
8.1.5 The '++' operator ... *47*
8.1.6 Taking into account other elements ... *47*
8.1.7 Assignments using the '++' operator ... *48*
8.1.8 Assignments using the '+++' operator ... *48*
8.1.9 "and" in filters, "or" in filters ... *49*
8.1.10 Testing a multiset ... *49*
8.1.11 Testing a sequence ... *49*
8.1.12 Testing a tuple ... *50*
8.1.13 Testing a dictionary ... *50*
8.1.14 Testing a structure ... *50*
8.1.15 Testing a referenced object ... *51*
8.2 THE 'RULE' ... 51
8.2.1 Simple Rules ... *52*
8.2.2 Multiple rules ... *52*

8.2.3 Extraction using rules......52
8.2.4 Using dictionaries and rules......53
8.3 EXERCISES......54
8.4 SUMMARY......54

9. ITERATIONS......55

9.1 BREAKING AN ITERATION LOOP......55
9.2 ORDER IS IMPORTANT......56
9.3 CREATING ITERATING PAMS......56
9.4 EXERCISES......57
9.5 SUMMARY......57

10. RECURSION......58

10.1 EXERCISES......58
10.2 SUMMARY......58

11. TESTING TYPES AT RUN-TIME......58

11.1 ACTION SYSTEMS......60
11.2 SUMMARY......61

12. FURTHER COMPOSITION OF PAMS......62

13. CONCURRENCY......63

13.1 THE PUT COMMAND......63
13.2 THE READ COMMAND......63
13.3 THE BREAD COMMAND......63
13.4 THE GET COMMAND......63
13.5 THE BGET COMMAND......63
13.6 DINING PHILOSOPHERS......64
13.6.1 Brief Explanation......65
13.7 SUMMARY......65

14. THE USE OF MODULES......66

14.1 THE IMPORT COMMAND......66
14.1.1 Renaming imported definitions: The 'as' keyword......67
14.1.2 Importing all definitions: The '' keyword......67*
14.2 SUMMARY......67

15. GLOSSARY......68

16. LIST OF CIRCUS-DTE KEYWORDS......69

16.1 LIST OF CIRCUS-DTE OPERATORS/SYMBOLS......69

17. INDEX......70

18. REFERENCES......72

19. ACKNOWLEDGEMENTS......72

20. APPENDIX A: SOLUTIONS TO THE EXERCISES......73

20.1 CIRCUS-DTE TYPES......73
20.2 CIRCUS-DTE VARIABLES......74
20.3 LAMBDA FUNCTIONS......76
20.4 POLYMORPHIC ABSTRACT MACHINES......77
20.5 ITERATIONS......77

1. Introduction to the Tutorial

1.1 Foreword

What is a good programming language? Nobody can really answer, because designing a language is somehow art and even if, sometimes, mathematics enables precise semantics and correct type systems they don't bring qualities like compactness, readability and expressiveness.

Most probably a programming language should be good for solving a particular category of problems by providing adapted programming constructs.

Circus-DTE has been designed around the paradigm of structure transformation, which is a rather new approach for modeling problems, but particularly suited for transformation related computations, e.g. language processor or transformation of structured documents.

The guiding principles for the design of Circus-DTE were:

- friendly but powerful type system
  (expression of structure schemes and static verification)

- minimal but expressive set of constructs
  (simplicity, legibility, flexibility, clarity)

- powerful "structural" pattern matching
  (three fundamental operations unified in one operator)

- composition operators
  (modularity and reusability of computational units)

Circus-DTE is also an attempt to find trade-off and to pull a language into the "magic" polygon where the whole is much more than the sum of the parts.

Jean-Yves Vion-Dury; designer of Circus-DTE.

1.2 What is Circus-DTE?

Circus-DTE is a primarily a formal, strongly typed language which has many strengths including strong rule based features and a powerful set of built in types, including MultiSets, Sequences and Dictionaries.

It is intended to be used primarily as a transformational language where there is a need to modify a structure from one type to another. The reason this language is well suited to this type of application is because we can strongly focus on rule based programming, combined with the ability to compose these rules in a structured method.

1.3 Purpose of this document

The purpose of this document is to introduce a person to the language of Circus-DTE, to demonstrate the key features of the language, and to explain some of the fundamental concepts. Wherever possible the code will be included in full and explained in detail, where appropriate.

The document begins by explaining the basic types of the language and moves through to more advanced features such as lambda functions, Rules, Pattern Matching and Polymorphic abstract machines.

Towards the end of most sections there is a small set of exercises that the reader may wish to try. It is recommended that these be attempted, to ensure that a feel for the language is obtained. Solutions to each of the exercises can be found in the appendix, and although these are not necessarily the optimal solutions, they serve the purpose to demonstrate how one might go about solving each problem.

1.4 Target Audience

Programmers who are just beginning to work with the Circus-DTE language or readers who would like to know more about the formation and compositional ideas that Circus-DTE provides.

1.5 Requirements

The reader is expected to have a general level of computing programming experience.
In this current version of Circus-DTE (as of August 1999) a **Unix** shell and a **Python** installation are required.

1.6 Circus-DTE Version

This document applies to the Circus-DTE language as of October 2002 and the Circus-DTE Compiler, version v2.0a.

1.7 Conventions

Below are a number of conventions used in this tutorial:

- Bold and Italic words are used to highlight ***keywords*** in the Circus-DTE programming language.
- Bold words in Courier font are used to show commands that you should type on the terminal. For example:

```
echo 'Circus-DTE is a strongly typed language!'
```

Output from the screen is shown like this (blue background, white text):

```
Circus-DTE is a strongly typed language!
```

- Bold words are used to show either **functions** or **types**.
- Code examples are shown like this (yellow background, black text):

```
module MyCode {

      <my code will appear here!>
}
```

- Italic words delimited within '<' and '>' signify something has been removed for clarity. For example <*snip*> would show a section of non-significant code or output has been removed to maintain readability.

- Often the module identifiers too, will be removed from the code, but the reader should be aware that the diagrams are code snippets, sometimes without necessarily having a need to "<snip>" the module identifiers from every code example.

Code or paragraphs within a Grey block is often a supplemental piece of information and is included more for completeness than for explanation purposes. Tips or additional comments about particular topics are highlighted within the Grey boxes.

This demonstrates a tip or additional information area:

Its use is to expand either in a different direction or provide additional information to a number of topics, but which may be skipped at the discretion of the reader.

2. Beginning Circus-DTE: Hello World!

The first thing that every programmer is shown when learning a new language is how to write a simple program to display a string to the screen. The tutorial begins with this example….

2.1 Step 1: Creating and editing the code

Create the following Circus-DTE file in any text editor.

```
module Test {

  // simple test to print a string

  const main:=pam i:[String*],o:Int
  .(
    printS("hello world!")
  )
}
```

2.2 Step 2. Saving your file

Save the file, ensuring to name it with the extension "**.ci**" – for example: **helloworld.ci**

2.3 Step 3. Compiling and Running the code

Compile the program, with the command:

```
cir helloworld.ci
```

(or, as the .ci is not compulsory…) `cir helloword`

This function compiles the program. We now have an executable (called with a shell generated script) which is named the same as our module, in this case Test. By running Test from the shell prompt we run the code.

```
./Test
```

This command executes our Circus-DTE program.

2.4 Step 4. Reading the output

Confirm the output looks something like this:

```
hello world!
```

2.5 Walkthrough

The code above works in the following way:

First we declare a **Module** called *'Test'* which contains our code.

Then we declare a **constant** of type **pam** called "main". A ***pam*** can be thought of as a function declaration – and in a similar style to C, a **pam** called 'main', in Circus-DTE, which accepts as input a sequence of Strings, returning an Int has a special operation – is our starting point of our program.

Within our main function we call the function '**printS'** (a built in function that displays the given variable to the screen).

Modules do not need to contain a main pam, but if they do not, an executable file will not be created (although they will still be compiled).

2.6 Troubleshooting

Check the code above looks like your file and try to see where the compiler is directing your attention, pay attention to the line numbers the compiler reports having problems with. Use the '-e' (*errors*) option to get more detailled error reports, and '-v' (*verbose*) option in order to track thecompiler behavior.

Try experimenting by changing some of the code, or omitting braces from the code to see the varying error messages you are shown.

2.7 Explanations

Given this program is working on your system, there now follows an explanation of the parts that make up this example:

The **Module** identifier:

```
module Test {

}
```

This is the encompassing module from which the code is run from. All code must reside within a module. See the **Module** Section for more details about how to effectively use modules.

A module name can be any word, starting with an alphabetical character, and must not contain spaces or symbols. You may include numerical characters in the module identifier, but these must appear after the first character of the module name.

The **constant** declaration:

```
const main:=pam i:[String*],o:Int
.(
  <snip>
)
```

Here we can see the constant *main* being initialised to a **pam** [Polymorphic Abstract Machine] (a kind of function). We are declaring that there is a pam whose name is **'main'**, and which takes a sequence of Strings as input (which we will refer to in this function as 'i', and returns the variable o, which is an integer.)

The code within the brackets is the code that is executed when this pam is called.

Finally, it can be seen that Circus-DTE allows the C++ style notation for comments, namely '//' or '/* … */' (although nested comments are not allowed)

2.8 Notes on the Compilation

Once compiled a **.pyo**, a Python file, is created with the name of the module (appended to an underscore). In our example, the file generated would be "**_Test.pyo**". This file, contains various information among which an intermediate representation of the programs and data defined in the module. Beside this main output file, other python modules (in the *helloworld* case, only one named"**_Test_0.pyo**") are generated for each functor (**pam** or **lambda** functions) encountered in the module.

Another file is also generated, which is an executable script, used to run the program. Its name is the name of the module (in our example, simply "**Test**").

2.8.1 Verbose output

It is possible to get a more detailed description of the actual compilation as it happens, by using the verbose flag –v, on the Circus-DTE compiler. For example:

```
cir –v <filename.ci>
```

2.8.2 Output with Error Reporting

Using the –e flag it is possible to view detailed error reports from the Circus-DTE compiler.

2.8.3 Options help for the compiler

Running the compiler with simply the: `cir` command should provide the following message:

```
Xerox Research Centre Europe

Circus-DTE compiler version 2.0a on /local (Wed Oct 13 17:05:28 MET DST 2002)
option -h gives information on usage
```

A full list of Circus-DTE compiler commands can be obtained using the –h flag.

```
cir -h
```

Some of the more useful commands to use with the compiler are as follows:

```
cir –e
```

The '-e' flag provides detailed error reporting.

```
cir –v
```

The '-v' flag provides more verbose error messages.

```
cir –list
```

The '-list' flag provides the list of all modules composing the compiler, together with versionning or location information. It also brings a useful performance mesure. On a PC-linux station (Pentium 4- 1.8 Ghz, local disk installation), it requires 1.1 seconds to execute the command.

Whilst compiling Circus-DTE programs it is a good idea to use '-v –e' flags to gain a better understanding of the errors or problems that may be encountered.

2.9 Summary

We have seen how to write and compile a simple Circus-DTE program, a simple demonstration of the print command and the function declarations using a **pam**.

We have also briefly looked at some of the Circus-DTE compiler options and seen how to run a compiled Circus-DTE program.

3. Circus-DTE Types

The basic element of any program is a variable. Each variable is of a particular **type**, for example a **String** or an Integer (**Int**). Circus-DTE represents these elements as either a variable (**var**) or a constant (**const**).

Constants are declared at the top level of any Circus-DTE module. Therefore constants are available throughout the whole *scope* of a Circus-DTE program. The section on variables discusses the ideas of scope and how to use the **var** command – this section discusses the relative types and shows how to create a **const** for that particular **type**.

Before beginning the section on types, we first look at a useful command, the ***verify*** keyword.

3.1 When to use the verify command

The '***verify'*** command is used mainly for testing purposes, to ensure that a given test condition has occurred. We often use it as a test of the variable or as calling mechanism for the print command, to obtain the useful side effect, so that one may compile and test code quickly – without having to define a main pam, etc.

The **verify** condition (example):

```
      const myString := ‘some’+A verify (myString=="something to test")
```

Here we can see the verify command which takes a **boolean** expression and evaluates it. Several verify command can be associated to one constant definition.

If all these expressions evaluate to **true** then the assignment or command which precedes the verify commands is said to be verified, which means that the definition is inserted into the output file. In any case, the compilation will continue. Expression evaluation has the syntax of every Circus-DTE boolean expression ; in that example, it uses a "double equals".

Verification is done at compile time. If the verify clause fails the check, the compiler will report an error, skip the generation of the associated constant, and continue the compilation.

If a ’verify’ test fails at compilation an error will be generated.

For example:

```
module Test {

   const badInt:="2" verify (badInt=="3")

}
```

Produces the following result:

```
/opt/exchange/Circus-DTEExamples/myTests/test.ci ...
"/opt/exchange/Circus-DTEExamples/myTests/test.ci", line 3: definition
"badInt" doesn't pass the verify clause
```

3.1.1 Testing with the verify command

The 'verify' command is very useful as a way of printing intermediate results as a simple form of debugging. For example, consider the following piece of code:

```
module Test {

   const test:="2" verify (print(test)==none)

}
```

In this example the 'verify' command checks to see if the output of the print command returns ***none.*** However, the print command always returns ***none*** – so what use is this verify command? It is useful because we use it to obtain a side effect. Upon compilation, the **print** command is actually executed, so we can see the printed value of our variable "test" as our code is compiled, on the display. There is no need to run the compiled code to obtain the print result, it actually happens when the code is compiled.

3.2 Base types

Circus-DTE offers :

- primitive types (such as Boolean, Integer, Float, String, Bytes) upon which it is possible to build new and more sophisticated types
- constructed types (such as Sequence, Structure, Dictionary) that are, by construct, made up of other types (eg. Sequence of Int)
- generic types that can be used in situations where either no particular type is required (Void) or, at the opposite, when any type could apply (Any)

These base types are presented below.

3.2.1 The Boolean type

A boolean (in Circus-DTE: **Bool**) can be either ***true***, ***false*** or ***none.*** Circus-DTE Booleans implement a 3-state logic where ***none*** means "unknown".

3-State Boolean **And** logic:

Op 1	Op 2	Result
False	False	False
False	True	False
True	False	False
True	True	True
True	None	None
None	True	None
False	None	False
None	False	False
None	None	None

3-State Boolean **Or** logic:

Op 1	Op 2	Result
False	False	False
False	True	True
True	False	True
True	True	True
True	None	True
None	True	True
False	None	None
None	False	None
None	None	None

Here is an example of defining simple booleans:

```
const b1 :Bool= true
const b2 :Bool= false
const b3 :Bool= none
```

Booleans can be the result of a comparison operation

```
const b4 := (10 > 2)
const b5 := (b4==true)
```

They may also be combined through usual boolean connectors

```
const b6 := false or (10>2) verify (b6==true)
const b7 := false and not (10>2) verify (b7==false)
```

Here we see how '**none'** is evaluated in operations

```
const b8  := (none and true)  verify (b8==none)
const b9  := (none and false) verify (b9==false)
const b10 := (none or true)  verify (b10==true)
const b11 := (true and none) verify (b11==none)
```

3.2.2 The Int and Float types

Numerical values come in two types, in Circus-DTE: ***Ints*** and ***Floats***. The bounds of the **Int** and **Float** are as follows: **Int** is unbounded – i.e. similar to Python's long value (to allow maximum flexibility for developers without needing to worry about restraints other than memory). **Floats** are mapped to the **IEEE Double** standard. In Circus-DTE, Int is considered a subtype of Float, meaning that **Int** values are also special **Float** values.

Below are some examples of defining Ints and Floats, and also our first look at using some built in functions.

```
const a :Int = 10
const x :Int = 5381791260    // or bigger!
const b := 10+a
const f := 10.3

// 'round' is built-in function that takes and returns a float, rounded
// to the nearest whole number

const f1 := round(10.6)

// A float and an Int can be compared, since they have a numerical
common type

const d:=10.1 verify (d > a)

// below is an example of the 'int' function which removes the fraction
// part of a float value and returns an Int.

const g:=10.8 verify (int(g)==10)
```

3.2.3 The String type

Strings are standard multi-character expressions, which may contain literal (unicode) characters (escaping defined with the '\' operator) and may be delimited with quotes or double quotes.

The operators '+' and '-' may also be used as concatenation and suppression operations.

String comparison is achieved with the operators '==' while the '<' and '<=' operators correspond to strict and non-strict inclusion.

Strings have the following operations available: card, addition, subtraction, multiplication, division and slicing.

```
const a  := "Circus-DTE"
const b1 := "cir"
const b2 := "cus"
const q  := "this is a test of 'quotes'"
const q2 := 'another test of "quotes" '
const c  := 'ci' + "rcus" verify (c==b1+b2)
const t  := '\''    // ie t is a single quote
```

Subtraction s1 - s2 is defined as the suppression of the first occurrence of s2 in s1. If s2 doesn't occur, s1 is left unchanged. For example:

```
const a  := "Circus-DTE"
const e := a - b1          verify (e=='cus')
const f := a - 'no-chance' verify (f==a)
```

Multiplication is defined for Strings, by example, in this way:

```
const m:=('Circus-DTE'*2)
const m1:=a+a
const m2:=(2*a)
const m3:=(a*2)
const isTrue:=((m==m1==m2==m3=='Circus-DTECircus-DTE'))
verify isTrue
```

Note that in Circus-DTE, `(m==m1==m2)` is exactly (and *computationaly*[1]) equivalent to `(m==m1) and (m1==m2)`.
Division is defined for Strings as executing the subtraction operation for as long as possible. (Notice that the multiplication/division operators are not symmetrical). Below is an example:

```
const g := "c i  r c   us" / " "  verify (g=="circus")
const h := "ci..rc..us" / ".." verify (h=="circus")
```

Slicing is the referencing of a sub-element by two indexes, inclusive of the first index, exclusive of the last. Indexes begin with 0.

Negative values indicate the index is referenced from the end of the variable, moving towards the front.

The **card** function returns the number of elements within a String.

```
const a:= 'circus'
const b:= a[2:3]        verify (b=='r')
const i := a[1:-1]      verify (i=='ircu')
const j1 := a[-3:-2]    verify (j1=='c')
const j2 := a[10:-2]    verify (j2=='')
const k := a[0:card(a)] verify (k==a)
```

It is also possible to omit one of the parameters to reference from the beginning or to the end of the String[2]. For example:

[1] As a consequence, (A==f(A)==B) will trigger 2 subsequent calls to the functor f.
[2] This is not yet implemented in the current version (0.3) of Circus-DTE

```
const l := a[2:] verify (l==’rcus’)
const m := a[:3] verify (m==’cir’)
```

String comparison is achieved with the operators ’==’.

The ’<’ and ’<=’ operators correspond to strict and non-strict inclusion, for all applicable types (String, Sequences, Multiset, Dictionary)

The ’<’ operator is rather different than standard string comparison where the lexicographic ordering is used. Any two Strings are totally ordered lexicographically, but it is not the case with strict inclusion: (’ab’ < ’cd’) and (’cd’ < ’ab’) both evaluate to ’**false**’ (and we cannot say that one string is "greater" than the other).

```
const a:=’ab’ verify (a<’abc’)
const b:=’b’  verify ((b<’b’)  ==false)
const c:=’cd’ verify ((c<=’c’) ==false)
const d:=’d’  verify (d<=’de’)
```

3.2.4 Bytes Type

The *Bytes* type is a primitive type that is used to handle strings of *bytes* instead of two-bytes unicode *characters*. Bytes strings are enclosed within a pair of opening single back-quotes with a space between each pair, e.g. ‘01 EF‘. They are especially useful as a pivot type in order to manage various string encoding format conversion or to handle binary files in memory.

The Bytes type has the same operations as String.

3.2.5 The Multiset type

A Multiset is the base collection type, containing elements of one or more types. They may contain multiple instances of the same value. They are defined within braces, i.e.: …:={1,2,3,4}. Order within Multisets is not significant.

Multisets have the following operations: card, addition, subtraction, multiplication and division. Multisets may also have the *inclusion* operators applied to them (see later)

```
type myMultisetType = {Int}
const myTest:myMultiSetType={1,12,13,1}
const howMany:=card(myTest) verify (howMany==4)

const strangeMuliset:={true,7,8.0,8.3,"this is legal"}
```

Addition is simply the appending (remembering that Multisets have no defined order) to the set, of a new set of values.

```
const myTest:myMultiSetType={1,12,13,1}
const add13Test:=myTest+{13} verify (noOnesTest=={12,13,13})
```

Subtraction is the removal of the specified set. If the set is not completely matched within the original set, only the values found are removed :

```
const myTest:myMultiSetType={1,12,13,1}
const no12Test:=myTest-{12} verify (no12Test=={1,1,13})
const no13Test:=myTest-{1,99} verify (no13Test=={1,12,13})
```

Multiplication operations may be applied to Multisets, in a similar way that we have seen for Strings, i.e. with an Integer:

```
const myTest:myMultiSetType={1,12,13,1}
const doubleTest:=myTest*2 verify (doubleTest=={1,1,12,12,13,13,1,1})
```

Multiset Division is accomplished by applying the division operator with a compatible multiset. For example:

```
const myTest:myMultiSetType={1,12,13,1}
const noOnesTest:=myTest/{1} verify (noOnesTest=={12,13})
```

Like String, Multisets may be tested with strict and non-strict inclusion operators (< and <=).

```
const a:={5,2,1,4,3}
const b:={3,4}  verify (b<a) and (b<=a)
const c:={1,2,3,4,5} verify ((c<a)==false ) and (c<=a)
```

3.2.6 The Sequence type

A *Sequence* is an ordered collection of elements of given type(s). A Sequence type is constructed by the '**[ ]**' (square brackets) symbol and a regular expression in between. A Sequence term is made up of elements separated by commas and multiple instances of the same element can be put in a Sequence (e.g. `[0,1,'2',3]`).

```
type type_name = [aRegularExpression]
```

where **aRegularExpression** is defined by the following grammar :

```
S ::= S*  |  S+  |  S? |  S{a,b} |  S '|' S'  | (S) |  S, S | S & S' | t
```

with :

- * meaning : 0 to many occurrences
- + meaning : 1 to many occurrences
- ? meaning : 0 or 1 occurrence
- {a,b} meaning : a to b occurrences
- '|' meaning : or
- ( ) showing priority
- , meaning : followed by
- S & S' equivalent to : (S, S') | (S', S)

- t : a type

As the order within a Sequence is significant, it is preserved for all operations performed on the sequence.

Examples

The following example is the definition of a Sequence of Integers type :

```
type mySequence = [Int*]
```

The example that follows indicates how to build a Sequence made up of Tuples that comprises an Integer and a String. It shows also how to set a constant for this type.

```
type mySequence = [<Int, String>*]

const V1: mySequence = [<10, "ab">, <-1, "C">, <705, "abc123">, <-123,
"123abc">]
```

The example that follows indicates how to build a Sequence made up of exactly one String followed by two Integers possibly followed by a Tuple that comprises an Integer and a String. It shows also how to set a constant for this type.

```
type mySequence = [String, Int{1,3}, <Int, String>?]
const V1: mySequence = ["xyw", 3, 4, <-123, "123abc">]
```

The Sequence type has the following operations available:

- Addition (concatenation)
- Subtraction (withdraw the first occurrence of a sub-sequence)
- Multiplication by an integer (concatenate itself n times)
- Multiplication by an other sequence (the ordered Cartesian product)
  [1,2]*[‘a’, ‘b’]==[<1, ‘a’>,<1, ‘b’],<2, ‘a’>,<2, ‘b’>]
- Division by a sequence (withdraw all occurrences of a sub-sequence)
- Slicing (return the sequence which is the part defined by indexes). Refer to section **Error! Reference source not found.** on page **Error! Bookmark not defined.**.
- card (return the number of contained elements)
- Inclusion test (check if sequences are, strictly or not, included)
- Scanning (using the for….do loop)

Note: It is not always possible to index entries in a sequence. To access a particular entry, the type must specify a deterministic number of entries. For instance, a variable v of type [Int*] cannot be accessed through v[0], because, it may have no elements whereas vv of type [Int+] can be accessed through vv[0] or vv[-1] since it will always have at least one element. A general way to access sequence entries is to either use slicing or a filter.

Demonstrated below is the creation of two sequences and the addition operator applied to them.

```
// define two Sequences, and a third being the result of the two
// concatenated

const s1:= [0,1,2,3]
const s2:= [10,9,8]       verify(card(s2)==3)
const s3:= s1+s2          verify(s3==[0,1,2,3,10,9,8])
```

Here we see the *subtraction* and *division* operations on a **Sequence**:

```
const k1:[Int*]= [1,2,3,2,3,4]
const k2:[Int*]= [2,3]
verify ( (k1-k2) == [1,2,3,4])
verify ( (k1/k2) == [1,4])
```

…and the *multiplication* operation:

```
const l:= [1,2,3,2,3,4] *2 verify (l==[1,2,3,2,3,4,1,2,3,2,3,4])
```

Sequences may also be sliced, using two indexes using the slicing operation. This operation returns a sequence. For example:

```
type NewSequence=[String*]
const mySequence:NewSequence=['1st element','2nd element','3rd element']
const slicedSeq:=mySequence[0:1] verify (slicedSeq==['1st element'])
```

As has been shown for strings, strict and non-strict inclusion maybe tested for within a Sequence.

```
const a:=[1,2,3,4,5]
const b:=[3,4]  verify (b<a) and (b<=a)
const c:=[1,2,3,4,5] verify ((c<a)==false ) and (c<=a)
```

3.2.7 The Tuple type

A tuple is an ordered set, or collection, of elements, which has a fixed size.

Tuples are defined with a comma delimited type list, encased within '<' and '>'.

Tuples have the ability to be referenced by an index value (delimited by square brackets: *see below*) and can be thought of as being similar to a structure (*see Structure Section*), but without field names for each of the elements.

It is not possible to either divide or multiply tuples, either with other tuples or with numerical values.

```
type MyTuple=<String, Float, Bool>

const tuple: MyTuple = <'This is a tuple',1.02, true>
const secondItem:=tuple[1]      verify(secondItem==1.02)
const lastItem:=tuple[2]        verify(lastItem==true)
```

It should be noticed that in the current version of Circus-DTE, tuples may not be sliced, however – this may change in the future.

Tuples have no addition or subtraction operators; therefore they also have no associated **card** function (as they are always the same length from the point of definition).

3.2.8 The Structure (Record))type

As in C, it is possible to define a structure that has a collection of named, typed elements contained.

As with other languages; structures may not be defined dynamically, and it is not possible to change the name or the type of the fields after compilation. Structures are delimited with the signs ('<','>'), and have *Variable: Type* pairs separated by commas.

To assign values to a structure, the use of '=' is required, in place of the ':' and the value follows.

In the example below we see the creation of a structured type (car). A constant of the type is then created (myCar) and then we access an element within the structure (the seats element):

```
type Car = <colour:String, seats:Int, engineSize:Float>

const myCar:Car= <colour='Black',seats=4,engineSize=1.8>
const myCarSeats := myCar.seats verify (myCarSeats==4)
```

3.2.9 The Dictionary type

The dictionary is a collection of elements that are each accessed by a key. Each key must be unique and the definition defines both the key and element types.

Dictionaries types are defined within '{' and '}' and take the format of `<key>` : `<object>` - where the key is a unique value and object is an element retrievable by the key.

Dictionaries have the following operators available: card, addition and subtraction – along with a number of built in functions. Dictionaries may also have the *inclusion* operators applied to them (see later)

To assign values to a dictionary, the use of '=' is required, in place of the ':' (in the type definition) and the value follows. For example:

```
type DictStringInt = {String:Int}
const myDict: DictStringInt={'uniqueKey'=1,'anotherUniqueKey'=2}
```

To add and remove elements to and from a dictionary, use the '+' and '-' operator respectively. For example, continuing from the example above:

```
const myDict2:=myDict+{'anotherKey'=3}
const myDict3:=myDict-{'uniqueKey'=1}
```

If you try to add an element that has the same key as an element already within the dictionary, the original element will be overwritten. Removing an element that does not exist in the dictionary has no effect.

The **card** function is also available for Dictionaries. Here we see it demonstrated:

```
type DictStringInt = {String:Int}
const myDict: DictStringInt={'uniqueKey'=1,'anotherUniqueKey'=2}
const howMany:=card(myDict) verify (howMany==2)
```

There are 3 built-in functions associated with dictionaries: **keys**, **items** and **values**.

Here is an example of each function in use (using myDict2 as defined above):

```
const myKeys:=keys(myDict2)
const myValues:=values(myDict2)
const myItems:=items(myDict2)
```

A Table listing what each constant is now set to:

Variable	Type	Value
myKeys	Multiset of Strings	{"uniqueKey","anotherUniqueKey","anotherKey"}
myValues	Multiset of Ints	{1,2,3}
myItems	Multiset of Tuples (Tuple type corresponding to the Dictionary type defined previously)	{<"uniqueKey",1>,<"anotherUniqueKey",2>,<"anotherKey",3>}

Had our dictionary of type DictStringInt (defined above), been defined differently, the corresponding changes would affect the types returned. For example, had our value field been defined as a **Float** then *myValues* would be of type {**Float**} (i.e. a Multiset of Floats).

Last, dictionary inclusion operators (strict and non-strict) behave in a similar way to multiset inclusion. Each value-key pair within the dictionary can be thought of as an individual element in a multiset.

```
const a:={'one'=1,'two'=2}
const b:={'one'=1}  verify (b<a) and (b<=a)
const c:={'two'=2,'one'=1} verify ((c<a)==false ) and (c<=a)
```

3.2.10 The Range Function

The *range* function is a function used to generate Sequences of either **Ints** or **Floats**. The function is a *lambda function* (see later) and may be supplied with up to three parameters. To obtain a sequence of Integers from the range function, the usage is as follows (use float parameters for getting a sequence of floats):

```
const mySequence := range( <step size>) (<start value>) (<end value>)
```

So, to obtain a **sequence** of odd Integers between 500 and 1000, the following command would be used.

```
const myOddInts := range (2)(501)(1000)
```

3.2.11 The None type

We have briefly seen the constant **none**, which is the only value in the type: **None**. The value 'none' can be returned when an answer is unknown or for example with the **print** function where the value returned is not significant.

3.2.12 The Void type

A type **Void** may be used which is useful for when we do not wish to pass parameters to a particular function. By declaring an input variable of a function as type **'Void'** we effectively say that we call the function with no parameters.

See the sections on **lambda** and **pam declarations** for more information as to the usage of the type **Void**.

3.2.13 The Unit Type

The Unit type is derived from the Bool type, but it is considered to be an abstract type that should not be instantiated by the programmer.

The Unit type has one value, **unit**, and it may be used in programming to signal to the compiler that the block evaluates to **unit**. Use of the Unit type is considered to be an advanced topic.

3.2.14 The Reference Type

The *Referenced types* are constructed types indicated by the keyword *ref* (for reference) *or wref* (writable reference). Note that **wref** is a subtype of **ref**. The writable references allow sharing and side-effects, whereas the read-only references just allow sharing. These last should be used as much as possible, since they provide quite safe mechanism to share data (and moreover, they offer a more flexible typing discipline).

```
type type_name = ref aType

type type_name = wref aType
```

Operations associated with the reference type are :

- the operator '@' is used for creating a writable reference on a given object
- the '!' is used to de-reference a referenced object
- writable references can be used to change derefenced objects:

Writable references can be used to change derefenced objects and create side-effects:

```
var a:wref Int = @ (10):Int
var b:ref  Int = a
.( (!a==!b==10) and ( !a += 1 ; (!a==!b==11))
```

Note also that (i) a type cast operation is required here, since the value 10 has type `Int in {10}`, and wref assignment requires strictly equal types ; (ii) if `!a +=1` is well typed, `!b += 1` would raise a type error.

Example : Trees having labeled nodes and Strings as leaves can be modeled by using references :

```
type tree =    <label : String , sub : [ref tree*]>
             | String
```

and an instance of such a type is the following tree T defined as :

```
var Tchildren : ref tree = @< label='body',sub=[@'test'] >
var T:tree = <label='html', sub=[Tchildren]>
```

Note that with the following definitions:

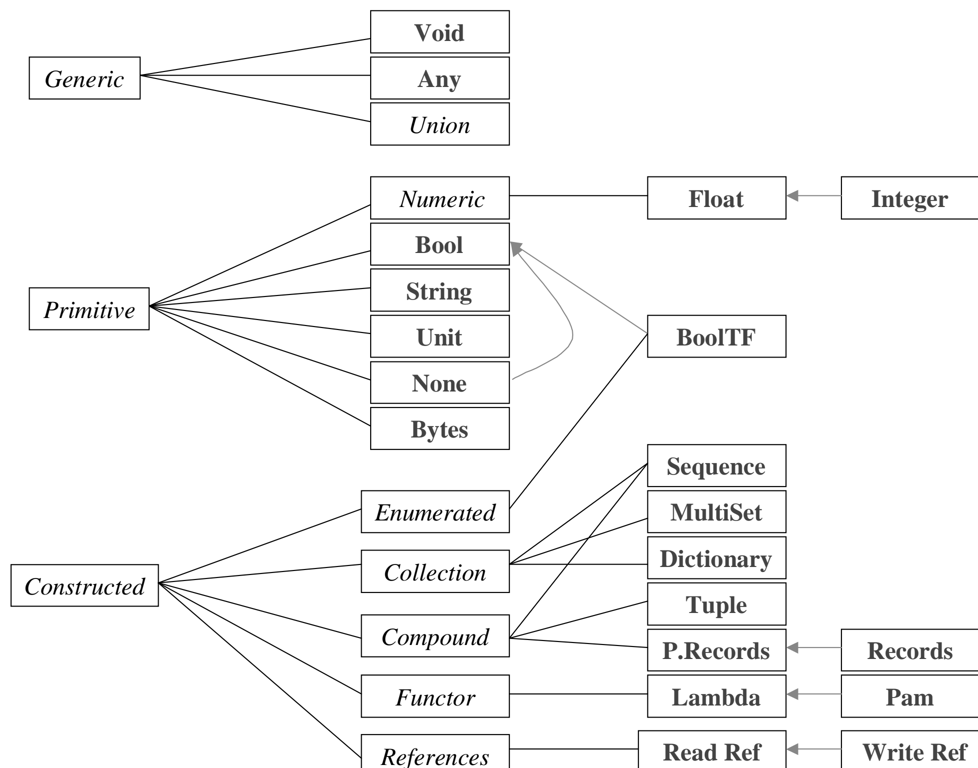
```
// anInt is a variable of type Integer
var anInt : Int
// A is a writing reference on the integer anInt
var A : wref Int = @ anInt
// B is a writing reference on the same integer anInt
var B : wref Int = @anInt
// C is a writing reference on an Integer and C is the same as A
var C: wref Int = A
.( (!A==!B) == true == (!A==!C) == (A==C) == (A!=B) )
```

the comparison (!A == !B) yields **true** as well as the comparison (!A == !C).

But the comparison (A==C) yields **true** while the comparison (A==B) yields **false**.

3.3 Polymorphism through Sub Types

The hierarchy of types in Circus-DTE is quite complex, as can be seen from the diagram below:



This shows that the type **Any** can be thought of as a super-type of each of the other types, whilst we also see that each type is a super-type of **Void**.

It is possible to enumerate each of the main basic types, and that **Ints**, **Floats** and **Bools** may be compared using *scalar logic* (i.e. we can test to see whether one Float is larger or smaller than another Float – to obtain either true or false). With types that are not scalar comparable, this is not possible, and the result could be true, false or none.

3.4 Type declaration

It is possible to declare a type in a similar way to constants.

You can declare a type based on one of Circus-DTE constructed types and on either one (or more) of the simple type(s) .

```
// sequence of integer
type seqOfInt = [Int*]

const mySeq:seqOfInt=[2,5,7,8]
```

The '|' Operator may be used to define the ability for a type to be of one type *or another* (multiple times). This construction is often referred to as a ***Union of Types***.

```
type seqOfStr = [String*]
type mySeq = seqOfInt | seqOfStr

const s1: mySeq = [1,2,3]
const    s2 : mySeq   =  [‘c’,’i’,’r’,’c’]
```

You can use enumeration to define a Type. An **Enumeration** is only used when defining **types** – unlike C++ and Java, and is a way to define a sub set of common elements.

As we see below, when defining a new boolean Type (**true** or **false**, removing the ***none***) **Enumerations** are used to define a subset of values to which the type may be assigned to.

The notation when defining a new *enumerated type* is as follows:

```
type <type name> = <Type> in {<element or complex expression>,...}
```

```
// define a boolean subtype with no 'none'

type BoolTF= Bool in {true,false}

// define a multiset enumeration of integers {0,1,2,3}

type myMultisetType = {Int in {0,1,2,3}}
const myTest:myMultiSetType={1,2,3,1}

// define a new type which could contain various types

type myType= Any in {'a string', 6, 8.88, false}
const myStrangeVar:myType='a string'

// below is an invalid assignment - this will not compile.

const fails:myType='oops'
```

You can also model recursive Types.

```
// define a recursive type to model trees of integers

type Tree= <node : Int> | <node : Int, sub : [Tree*]>

const myTree : Tree = < node = 0,
                        sub = [ <node = 1>, <node = 2>]
                      >
```

3.5 Exercises

For solutions to these exercises, see Appendix A.

1) Define a Type, *EvenIntsLTTen,* that may only contain even Integers, which are less than 10, which are stored in no particular order.

2) Define a simple date structure which is suitable for containing dates in the format dd/mm/yyyy

3) Define a Type, *Contact*, based on a structure that represents a contact found in a typical address book, able to store names, birthdays, telephone numbers and an address.

4) Define a Type, *ContactWithFullName*, which contains this structure, above, with a unique full name as a pair – with the principal idea of being able to retrieve a person's telephone number by their full name.

3.6 Summary

In this section we have seen the Circus-DTE types, **Ints**, **Floats**, **Bools**, **Strings**, **Multisets**, **Sequences**, **Tuples, Structures** , **Dictionaries** and **References** along with their sub-type relationships.

We have also looked at some operators on these types; along with *multiplication*, *subtraction*, *slicing*, *division* and *addition*. We briefly looked at the range function, used for generating **Sequences**.

We have seen :

- the **Void** type that may be used as a placeholder for when a function does not require an input variable
- the **Unit** Type that contains one value, **unit**, which is returned by assignments to the Circus-DTE compiler

We have introduced general mechanisms (union types, enumeration, recursive types) for creating your own types based on Circus-DTE base types.

4. Circus-DTE Variables:

Variables are used in Circus-DTE to store variable data that can be calculated from functions or complex expressions.

The general format is to use the '**var'** keyword followed by the name of the variable, followed by '.(' to open of a new scope, whereby this current variable now exists. End of scope is marked with ')'.

It is not possible to exclude the variable type, unlike a **const** variable, each and every declaration of a **var** must include the type information.

So, to declare a variable **String**, named myString and initialise it to a given value:

```
// variables can not be declared at the top level of a module!

var myString : String ='This is a string'.( <snip> )
```

Notice the full stop and opening bracket at the end of the definition. What follows (the code that has been <snipped>) is the new scope that now contains this variable which, for example could be another variable declaration and a further scope, or it could be a complex expression. A closing bracket ')' marks the end of the scope.

If several variables are defined successively, the '.(<snip>) can be written once for all these variables :

```
// declaring two variables in the pam myPam
const myPam := pam x : String, y : String.(
      var myString : String.(
      var myInt : Int .( <snip> )))
```

is equivalent to :

```
// declaring two variables in the pam myPam
const myPam := pam x : String, y : String
      var myString : String
      var myInt : Int .( <snip> )
```

No variables may be declared at the top level of a module. This means it is not possible to define global variables, and that they must be declared within the code of either a Pam, lambda function or even a verify clause.

Variables must be declared before being used except for :
- variable used in a filter (see 8. Pattern matching) for which declaration is not compulsory
- variable used in a For … do loop (see 6. Expanding the computational statements)

4.1.1 The differences between consts and vars

- **const**s may only be declared at the "top level" (i.e. global declarations) of a module – i.e. not in any function declarations, **vars** may not be declared at the top level, but only within code blocks such as **pams**, **lambdas** or expressions.

- **const**s may never change once declared, **vars** may change freely.

- **consts** do not require any type information upon declaration, **vars** require the explicit declaration of a **type**.

4.1.2 Scope of variables

Each variable declaration opens a new scope with the '.(' symbol. The scope of each new variable contains all the previously defined **vars** and **consts** in the current procedure.

Variable declaration may only be done inside a procedural block of code. It is not possible to define variables at the *top level* definition, as one is able to in the case of Constants. This means it is valid to declare variables within Pams, Lambda functions or in any actual *expression* (a *verify* clause, for example).

So, the following code is not allowed, as we are trying to define a global **variable**, which as we discussed earlier, do not exist in Circus-DTE, (i.e. a variable whose only scope is the module itself)

```
module WillNotCompile{

var myString :String= "This doesn't work"

}
```

But this example is valid:

```
module Compiles{

const myConst:String="Variables can be declared within a verify clause!"
      verify(
         var myString:String="This works fine".(
         print(myString)==none)
      )

}
```

In the example above, we notice that the print statement occurs within the scope of the variable declaration. This means that we are able to use the variable myString.

Here we can see an example of declaring several variables of some of the Circus-DTE types, all within the scope of the previous variable:

```
<snip>
var myInt:Int = 4
var myFloat:Float = 3.8
var myString:String = "hello"
var myMultset:{Int}= {4,9,10,4}
var mySequence[Int*]:= [9,8,7]
var myDictionary{String:Int}= {"a unique key"=25,"another key"=44}
var myBool:Bool= false.(
<snip>
)
```

4.1.3 Basic Built in functions

Some useful built-in functions will also be demonstrated in this section, below are a list of these.

Each function is described using the following format

Function name heading **Function name Input Type -> Output Type** **Function description.**

4.1.3.1 The chr Function

chr Int -> String
Returns the (unicode) character associated with the input **Integer** (or empty string '' if the input is invalid)

4.1.3.2 The ord function

ord String -> Int
Returns the **Int (*unicode*)** value of the first character in the **String**. (returns –1 if there is more than one character in the **String**) ['a' == chr(ord('a'))]

4.1.3.3 The str function

str Any -> String
Returns a **String** representation of the input variable. This variable may be of ***Any*** type. It also handles references and cycles

4.1.3.4 The round function

round Float -> Int
Returns the **Float** value, rounded to the nearest **Int** of the input variable. i.e. (**round**(4.5)==5.0), (**round** (4.49)==4.0)

4.1.4 The mod function

mod Int, Int -> Int
Returns the modulus of the first parameter, divided by the second. It is possible (as with all multiple parameter lambda functions) to pass in less than the required number of parameters, to obtain a new function which will calculate the modulus of the original parameter with each new parameter passed to it (see the lambda section for more details about multiple parameter lambda functions)

4.1.4.1 The int function

int Float -> Int
Returns the **Int** value of the **Float** which is passed to it (calculated by removing the trailing fraction).

4.1.4.2 The float function

float Int -> Float

Returns the **Float** value of the **Int** which is passed to it. Note that it's not required to use this conversion, (but it might be useful for programmers used to this kind of conversion). Indeed, Integer are considered as specialized floats in Circus-DTE, and thus do not need explicit conversions.

4.1.4.3 The card function

card **Any** **->** **Int**
Returns the number of elements in the variable (for example the length of a **String**, the number of values in a **MultiSet, Sequence** or **Dictionary**). The function ***'card'*** takes a *Union* of types, which is of the following types: String, [Any*], {Any}, {Any:Any}.

4.1.5 Randomising Functions

There are two main functions used in Circus-DTE, for generating random values:

4.1.5.1 The rand function

rand **void** **->** **Float**
Returns a Float value between 0.0 and 1.0

4.1.5.2 The choice function

choice [**Any***] -> [**Any?]** | {**Any**} -> {**Any}**
Returns a a value picked randomly from the Multiset or Sequence passed to the function.

4.1.6 Using the Built in functions

The code below shows examples of some of the above built in functions

```
// demonstrating the ord, float and chr functions.

const char:String='a'    verify (ord(char)==97)
const ordOfA:String=97   verify (chr(ordfOfA)=='a')
const i:Int=7            verify (float(i)==7.0) and (i==7.0)

// demonstrating the card function.

const tStr:='abc' verify (card(tStr)==3)
const tSet:='[1,2,8,10] verify (card(tSet)==4)

// demonstrating the randomising functions.

const a:= [1,2,3]
   verify var v:[Any?]=choice(a).( (v==[1]) or (v==[2]) or (v==[3]) )

const b:= {4,'5',6}
   verify var v:{Any}=choice(b).((v=={4}) or (v=={'5'}) or (v=={6}) )


<snip>
var c:Float=rand() .( (c<=1.0) and (c>=0.0) )
```

4.1.7 Casting Variables[3]

Typecasting of an expression e into a type t is achieved by the syntax '(e):t' but the casting operator is meant mainly for writing references and structured types, rather than basic Int -> Float operations, etc
So, a basic example of casting a Sequence of Ints into a Multiset of Floats would look like the following:

```
      var i:[Int*]=[1,2,3,4,5,6,7,8,9,10]
      var f:{float}=(i):{float}
```

4.1.8 Default Values:

The *Circus-DTE* compiler does not expect you to provide a value for each and every instance of a variable you create. The following table shows a list of the default values assigned to each variable, if one is not assigned at the time of definition along with the code required in order to be able to reset the variable to its initial default variable:

[3] Not yet implemented in the current version of Circus-DTE (0.3)

Circus-DTE Type	Example	Defaults to
Int	Int	0
Float	Float	0.0
Bool	Bool	false
String	String	''
Multisets	{t}	{}
Dictionaries	{t=t'}	{=}
References	wref t	@v v being the default value of the associated type t
Sequences	[S]	Depend on the expression S (try to minimize)
Tuple	<t_1,…,t_k>	Each member defaults to t_i 's default value: i.e. a tuple: **<String, Bool>** Defaults to: **<'',false>**
Structures	<…,m_i:t_k,…>	*Dependent on the defined structure:* Each member defaults to it's own default value: i.e. a structure: **<m1:Int,m2:[String*]>** Defaults to: <m1=0,m2=['']>

4.2 Exercises

1) Write a simple Circus-DTE program that proves that the length of an empty string is equal to the value of a float variable, set to its default value.

2) Create a constant, *NameList*, which holds a sequence of names (Strings) and have the program print out the first and last names, regardless of how many are in the sequence (assume at least one name is in the sequence)

3) Create 3 constants that represent possible values for the previous Contact type (see previous exercise) and then insert these into a new constant which stores them in the correct fashion, for the *ContactWithFullName* Type, also defined in the last exercise.

4) Create a constant that is assigned a random name from the *ContactWithFullNameList* constant defined earlier and print the phone number associated with this one from the same variable.

4.3 Summary

We have seen how to declare variables of each of the basic types, and also with our own defined types, generated from both enumerations and basic variables.

In Circus-DTE it is possible to cast variables to other types with the keyword '**to'**, although this operation should be reserved for structured data, which in turn maybe used as highly complex transformations.

Circus-DTE comes complete with some useful functions, such as **ord**(), **card**(), **str**() and **chr**() and these have been demonstrated.

We have also seen two types of randomising functions, **rand**() and **choice()** which offer two different methods of generating random values.

Finally, we have examined the default values for each of the types documented.

5. Lambda Functions

Lambda Functions are functions that take one input parameter, (a variable of one particular type), and return another variable that is derived from their core functionality.

Lambda functions are powerful in that they can be used to define specific actions for given types and they may be constructed in such a way that they can be decomposed in order to be able to provide additional functionality. These advantages will be discussed below.

There is no requirement to define the type of the return value of a lambda function as this is determined automatically.

5.1 Defining a lambda function

lambda <*input variable*> **:** <*input type*> **.** (<core expression>)

5.2 Using lambda functions

To define a constant, which is a lambda function, that increments an Integer, we implement the following code:

```
const lambdaInc:=lambda x:Int.(x+1)
```

5.3 Calling a lambda function

Here we can see that we simply pass into the lambda function (now defined as a constant, as in the code above) to obtain the output which we can use.

```
const myInt:=lambdaInc(5)      verify (myInt==6)
```

It is important to note that within the core functionality of a lambda function, it is not possible to use imperative expressions (those return a **unit type**). For example s := s+1.

Commands used in this core code must, in effect, be just one expression. In order to use imperative expressions, in your code, see the section on **Pams**.

It is, however, possible within the core functionality of a lambda function, to assign variables and constants, but the core may be composed of one and only one expression (as complex as needed).

For example:

```
const mylambda:=lambda x:Int
     .(
       var y:Int=lambdaInc(x).(x+y )
     )
//where lambdaInc is defined as above
```

Here we create a variable y within the core functionality and assign it to a function which returns a value given the input of x (in this case x+1). We finally return the final value of x + y.

To test this function:

```
const test :=mylambda(3) verify (test==7) // i.e. 3+(3+1)
```

5.4 Lambda Programming

As can be seen from the above example the lambda core functionality code must consist of just one expression. This expression may include (multiple) variable declarations, and other calls to lambda functions, but it must remain as just one expression. The value derived from the code is the value that is returned by the lambda function.

5.5 Further Examples

5.5.1 A 'HelloWorld' lambda function

```
// our lambda definition

const helloworld := lambda x:Void.(print("Hello World"))

// and our call to run this function

const myTest:=helloworld()
```

Note that as we have declared x as type **Void**, we do not need to pass any input variable to 'helloworld' in order for it to function correctly.

The variable 'myTest' will actually evaluate to "none" as this is the value derived from the print("Hello World") function.

5.5.2 Additional Parameters

It is possible to design a lambda function that takes two, or more parameters to derive one final result. This is achieved by specifying the expression of the core as another lambda function, which in turn may again specify a lambda function in its core.

The multiplication of two integers could be converted to a lambda function, which accepts two Ints.

Also in this example we define the logical expression for ***implies***; which may be noted as being the result of (*y* or not *x*).

```
module Logic {

  // lambda multiply
  const lambdaMultiply := lambda x:Int lambda y:Int.(y * x)
  const testMult:=lambdaMultiply(3)(5) verify(testMult==15)

  // lambda implies
  // type BoolTF is a builtin type defined as Bool in {true,false}
  const implies := lambda x:BoolTF lambda y:BoolTF.(y or not x )

  const impTT := implies(true)(true)  verify (impTT == true)
  const impTF := implies(true)(false) verify (impTF == false)

}
```

5.5.2.1 Extending the use of multiple parameter lambda functions

As we have seen above, it is possible to write a lambda function that takes two (or more) parameters, by supplying an additional lambda function within the core functionality of the output. As was demonstrated, by passing the required number of parameters to these functions we are able to satisfy both input requirements and produce an output. The question remains however, what are we returned, if we only pass to this function, one parameter (or any number of parameters less that the required amount)?

The answer can be found by examining the definition of our original lambda function – the *lambda multiply*:

```
     const lambdaMultiply := lambda x:Int lambda y:Int.(y * x)
```

We assume now we simply pass it one variable, and assign the result to a **const**, i.e.:

```
     const ourTest := lamdbdaMultiply(3)
```

By examining the original lambda function, and extracting out the known variable parts; we see that the ourTest constant is actually equal to:

```
                          lambda y:Int.(y * 3)
```

So in effect we have been able to isolate a particular lambda function, with a set variable. We would be in a position to utilise this by re-naming the constant 'ourTest' to 'multiplyByThree' and using it like any other lambda function: i.e.

```
     const multiplyByThree := lambdaMultiply(3)
     const myInt := multiplyByThree(6)  verify (myInt == 18)
```

We have, in effect, derived a new lambda function from a previous one, without modifying the original function. This could have many uses where sub sets of functions maybe created from more complex, previously written, lambda functions.

5.5.3 Anonymous functions

A lambda function may be defined anonymously by using its signature within a new type as a way of not having to define the function exactly.

For example a structure may contain an **Int** and a lambda function which dictates that it should accept and return an Integer. We could define this as follows:

```
type AnonTest=<m1:Int,m2:Int ->Int>
```

We would then be able to create a constant of this type and populate the anonymous function:

```
const myAnonTest:AnonTest=<m1=10,m2=lambda x:Int.x*x>
verify
var test:Int = myAnonTest.m2(myAnonTest.m1) .( (test==100) )
```

5.6 Exercises

1) Write a lambda function that returns the square of any integer or float that is passed to it.

2) Write a lambda function which calculates and returns the answers to the following expression: (x*3)+(y*2)+(z)

3) Without writing a *new* lambda function, but by extending the previous example, create a lambda function which calculates: 12+(y*2)+(z)

5.7 Summary

Lambda functions are composed of an input value and a piece of code which when computed derives the output. This code consists of 1 expression.

It is possible to assign lambda functions to constants and to reference these constants with parameter calls.

By returning a lambda function, instead of an actual value, it is possible to derive new functions which have less parameters, whilst retaining the original characteristics of the initial lambda function, without modifying it at all.

By extending the lambda output to be a new lambda function, it is possible for the lambda function to take additional parameters.

6. Expanding the Computational statements

So far we have seen how to define variables, types, *pams* and *lambda* functions; and we have looked at basic expressions and assignments. However, the strength of Circus-DTE is in the pattern matching, rules code and looping (iteration) constructs.

6.1 The for .. do loop

The construction for a 'for… do' loop is as follows:

```
for <variable declaration> in <container> do <expression>
```

Notice also that the 'for loop' construct also creates the iteration variable and handles the incrementing for the programmer already.

A container may be either a Multiset or a Sequence, whilst the variable declared will be automatically typed to the same type that is within the container. A container may also be a Dictionary of type {typeOfKey : typeOfValue}, whilst the variable will be of type <typeOfKey, typeOfValue>

For example:

```
const forloop:=lambda container:[Int*].
  for ourVariable in container do
     (print(ourVariable))

const test:=forloop([1,2,3,4,5,6,7,8,9,10])
```

This code demonstrates the creation of a lambda function which takes a parameter which is a sequence of Ints and iterates through them printing each element, using a 'for do' loop.

The code produces this result:

```
forLoops.ci ...

1
2
3
4
5
6
7
8
9
10
```

This simple and most often used construction of the 'for…do' loop is only a special case of the more general construction allowed in Circus-DTE,

```
for <filter> in <container> do <expression>
```

For example :

```
const forloop := lambda container:[<String,String>*].(
      for <?eachName , ?eachCountry> in container do
            (printS(eachName + ‘ comes from ‘+ eachCountry)))
const test :=
forloop([<’Andre’,’Hungary’>,<’Marian’,’England’>,<’Leon’,’Russia’>,<’Ta
nia’,’Russia’>,<’Luc’,’France’>])
```

The above code creates a **lambda** function that takes a parameter that is a Sequence of <**String,String**> and iterates through them. It then prints a sentence including the two components of each element using the ’`for...do`’ loop.

The code above produces this output:

```
forLoops.ci ...

Andre comes from Hungary
Marian comes from England
Leon comes from Russia
Tania comes from Russia
Luc comes from France
```

6.2 If, then, else

As in most programming languages, Circus-DTE includes the **if**, **then**, **else** construct.

The format is as follows:

```
if (<expression>)

       then
            <statements to run if the expression == true>
       else
            <statements to run if the expression != true >
```

It should be noted that if there is no **else** condition, the **if/then** statement must be expressed as a rule (it is not possible to have only an **if**/**then** statement). See later for a complete description on rules.

6.3 The’ [|…|]’ operator (or Action system)

The **‘[|…|]’** operator (or action system) is used to enclose a block of code containing a series of rules (see 8.2. The ‘rule’) and expressions separated by commas (,). The code is evaluated sequentially. The evalutation stops and a result is given as soon as an expression evaluates to **true**, the remaining items are not tested. If no such value is found then the end result is **none**.

Note : The ‘[|…|]’ operator is presented in more details in paragraph 11.1 .

```
// action system to simulate a psychotherapist
[|
sInput # %“I feel depressed” => sOutput := “Since when do you feel
depressed ?”,
sInput # %“I feel very happy” => sOutput := “Why do you feel happy?”,
sOutput := “Tell me about your childhood.”
|]
```

Note the equivalence between :

```
 if (<expression>)
      then <statements to run if the expression  == true>
      else <statements to run if the expression  ! = true>
```

and :

```
[|
(<expression>) =>  <statements to run if the expression  == true> ,
<statements to run if the expression  ! = true>
|]
```

6.4 The ‘map’ operator

The ‘map’ operator has two parameters, one is a lambda or a pam and the other is a sequence or a multiset. The result is a sequence (resp. a multiset) the elements of which are the result of applying the lambda or the pam to each element of the input sequence (resp. multiset).

```
const mySeq: [Float*]=[1.10,2.49,3.60]
verify(map(round)(mySeq) == [1,2,4])
```

6.5 Summary

'*For loops’* are constructed by using a new variable and the test to see whether it is 'in' a given container, and if so 'do' the subsequent actions. The construction of the testing variable and the incrementing of this variable are handled by Circus-DTE, unlike both C and Java.

The standard 'if then else' code block is available to the programmer, but not just the 'if – then' condition. If the programmer wishes to implement an if/then condition (with no else) then they should implement the condition as a rule.

The '[| …|]' operator contains a block of code to be performed as a controlled sequence of events, each separated by a comma. The comma is separating each rule and will act as a "*switch*", so if the first rule is executed (because the pattern match evaluated to **true**) then the remaining items, after the comma will not be tested. This continues until either the end of the block has been reached, or one expression has been evaluated to **unit**. In this case, **none** is returned.

The *‘map’* operator allows one to apply a function or a pam to each element of either a sequence or of a multiset.

7. Polymorphic Abstract Machines: (PAMs)

A **Pam** is the basic, extendable, unit of computation. As with *lambda* functions it has an input variable but it also has a typed output variable. It may also have a set of local variables and computational statements.

7.1 An Example Pam

```
const setStringPam:=pam src:Void,outp:String
 .(
  outp:='Hello World'
 )

const myPamTest:=setStringPam() verify (myPamTest=='Hello World')
```

7.2 Definition

To define a **pam** the following convention is used:

```
pam <input variable>:<type>, <output variable> <type>.( <output generation code> )
```

This may be used to assign to a **const** value, or used anonymously, like lambda functions.

7.3 Usage

This example shows a pam that computes and prints the ordinal value of a string's first character, along with this character. The value, a structure (defined with `<value:Int,char:String>`) is returned as the output type.

```
module Test {

const stringVal:=pam src:String,outp:<value:Int,char:String>
        var i: Int.(
        i:=ord(src);
        outp.value:= i;
        outp.char := src[0:1];
        print(outp)
     )

const testSc:=stringVal('a')

}
```

The result…

```
<value=97,char='a'>
```

Notice how we use ';' for the separators within our code segment, yet '.' for our variable separators. Each '.' is in effect creating a new scope within the declaration. They have the same effect as braces, in languages like C++, whereby a new scope is created, each with it's own accessible variables. In the above example

we create a new scope after defining the variable 'i'. This scope inherently contains the predefined variable 'i' as a type of global variable.

It should also be noted that we do not terminate the last line of the code segment, within the pam, with either a full stop, or semicolon. This signifies the end of the code block.

Semi Colons are used to separate two *imperative statements*, hence the reason why we also do not use them at the end of a code block.

For readability purposes, it is possible, but not a requirement to delimit the code block with parenthesises.

7.4 *Why* 'Polymorphic' *abstract machines?*

Polymorphism promotes the idea that pams are used in Circus-DTE, as building blocks and that the composition of basic elements yields greater benefit than the sum of the individual parts. The smaller and tighter the code of each pam, the more reusable that pam becomes.

This is of course an ideal situation. It is perfectly possible to write and use a pam that only deals with one particular type of variable.

To achieve pure polymorphism pams should accept a type of Any and perform run time type checking on the input variable to determine the code which should run on this variable. The section: **'Checking variables at run time'** deals with this procedure.

7.5 *Parameter passing, by* Copy

Values are passed by *copy*, rather than by *reference*, which means that it is not possible to update variables outside the scope of the code they are written in.

For example – this code demonstrates that it is not possible to increment the variable '*s*', using a newly defined pamInc function (use explicit writable references for that !):

```
const pamInc:=pam x:Int,out:Int
       .(
       x := x + 1;
       out := x
       )

const s := 4
const myInt : Int = pamInc(s)        verify (s==4) and (myInt==5)
```

We see here that the value of myInt is indeed 5, but 's' itself has remained unchanged.

7.6 *Composing pams*

This section is excluded from the current distribution

7.7 *Summary*

Pams are the basic building blocks by which most Circus-DTE programs will achieve their functionality. It is possible to include variables, types and computational statements within their code blocks.

The idea is that a pam takes in one parameter and outputs one parameter.

The *polymorphic* analogy is that one pam should be able to handle many different input and output types in as simple and efficient way as possible.

Parameters to the pam functions are passed by copy, therefore it is not possible to modify values of objects inside the scope of the new function, and expect them to be changed upon the exit of the function.

8. Pattern-Matching

A pattern matching system is made up of three parts: a **Subject**, a **filter** to match against and a **Rule** to execute, should the pattern be matched to the filter.

The operation '#' is used to complete pattern matching operations, and is usually followed by either check for containment (i.e. does an element match the filter) and often an assignment to another variable.

Pattern matching is a form of rule based evaluation, which is used in place of if/then statements commonly seen in other languages. It is a very powerful way to evaluate expressions and to program rule based systems.

8.1 The Subject

The Subject is defined as the variable to check the filter against. Subjects are commonly variables or expressions, but may include operators such as concatenation or functions that are computed.

Examples of a *Subject*:

- 'abcdefg'
- a **const** previously defined
- (a * 2)…
- myTestPam("x") …etc

8.1.1 Matching with a Filter

```
<subject> # <filter>
```

8.1.2 The '%' Filter

To test to see whether the string 'Circus-DTE' is equal to the string cString, the following statement is used:

```
cString# %'Circus-DTE'
```

The '%' operator is used as check to determine if a pattern has occurred, so the above line is checking cString for equality of 'Circus-DTE'. This rule is evaluated as successful if cString =='Circus-DTE'. For a pattern based approach for sub-strings, see below.

If cString had been a Multiset, Tuple, Dictionary, etc, we could also check for particular elements to be contained within the original variable.

To check to see if the String '*Circus-DTE*' appears in a Multiset, assuming we have a variable called cMultiset:

```
cMultiset# {%'Circus-DTE'}
```

i.e. where cMultiset == {'Circus-DTE'}; the rule evaluates to True.

Circus-DTE does not restrict the type of expression that can be checked, so it is possible to reference variables, evaluate complex expressions or constants within the area used to check for inclusion. For example:

```
cString# %('Cir'+'cus')
```

```
(cFloat*3)# %(myFloatFunction(2.135))
```

These are both valid pattern matching constructions – although each one does not have the associated *rule* alongside. See below for details about the *rule* constructs.

8.1.3 The '?' Filter

The question mark operator is used as a filter that is defined as "match any element". It may be combined with a variable (see later) or used in conjunction with the concatenation operator ('++') or on its own.

The following example shows how it is possible to match any tuple which only has two elements:

```
myTuple # <?,?>
```

The '?' operator can also be associated with a type to allow type checking in a filter. For example, the following filter will succeed if *whatIsIt* is of type Int (or of a type that is a subtype of Int) :

```
whatIsIt # ?:Int
```

8.1.4 Assignment with a Filter

`variable1 #  ?variable2`

This statement performs :

- a check of type compatibility between variable1 and variable2, then
- assignment (`variable2 := variable1`), if variable2 is of the same type as variable1.

See below in the section entitled "*Determining type equality at Run-time*" for a more detailed look at this topic.

If this statement is evaluated to true, we then are able to proceed, making use of variable2.

A variable *myVariable* used in a filter, can be a declared variable of type T (in particular, it can be declared as a global variable).

```
lambda x: Int|String
var myVariable : Int.( x  #  ?myVariable => myVariable*2 )
```

In this case, a check of type compatibility between the subject expression *x* and *myVariable* is performed at runtime, as a filtering condition prior to change the value of *myVariable*.

But one can also use a variable *myVariable* in a filter without declaring it before . The variable's scope is then the right hand side of the rule.

When using an undeclared variable *myVariable* in a filter, you can specify its type T. In the example below, type compatibility between variable1 and *subject* is performed. Again, *myVariable* will be assigned the result of *subject* only if *this one* is of type T (or of a type that is a subtype of T).

```
<subject> # ?myVariable:T => <snip>
```

Note also that pure runtime type check can be performed by

```
<subject> # ?:T => <snip>
```

When using an undeclared variable *myVariable* in a filter, you do not need to specify its as illustrated in the following example. Then, no type checking is performed. The compiler will statically infer a type for *myVariable* at compile-time.

```
<subject> # ?myVariable => <snip>
```

8.1.5 The '++' operator

We have already seen the concatenation operator, the '+' symbol - this operator combines elements (such as strings) together. We can think of the String "Circus-DTE" to be made up for 3 sub-strings (or many other sub-string combinations).

'Cir' + **'c'** + **'us'**

If we now focus on the letter 'c' in the middle of the string, we see that it can be made up of a 3-part expression:

?++ 'c' ++?

This is saying that we take an element and can see it being made up as "a filter of zero or more elements", plus the letter 'c', and a further zero or more elements.

With this in mind, we now see how to approach pattern-matching of elements which are "nested" within other elements.

8.1.6 Taking into account other elements

```
<subject> #     % <expression> ++ ?x
```

If the beginning of the subject matches the expression and continues with zero or more elements, assign the rest to x

```
<subject> #   ? ++ % <expression>
```

Succeed if the subject matches the expression with zero or more elements preceding the matched item

```
<variable> #   ? ++ % <expression> ++ ?
```

If the variable matches the expression whilst both proceeding and continuing with zero or more elements (for example, characters in a string) which can be used for sub-string matching.

```
cString# %('Cir') ++ ?
```

This example, above, evaluates whether a sub-string 'Cir' is the first 3 characters of cString. In this example it would not be possible to test for a sub-string of 'cus' (if cString=='Circus-DTE') and expect the evaluation to be '**true**'.

```
cMultiset# %{'Circus-DTE'} ++ ?
cMultiset# ?++ %{'Circus-DTE'}
```

This example shows that, as with Dictionaries, Multisets have no predefined order. The two expressions are equivalent.

Therefore for types with no defined order we can say that:

```
?++ %{e1}  •  %{e1} ++?
```

8.1.7 Assignments using the '++' operator

We have already seen previously that at every point where an expression uses a '?' we are able to append a variable after the '?' to make an assignment. This allows a programmer to assign a variable to the other elements within a given matched expression.

So for example given the variable:

V= 'Cir' + 'c' + 'us'

And the following filter :

V # ? v1++ % 'c' ++? v2

…we get : **v1 == 'Cir' and v2=='us'**.

So, with the use of a variable and the '**?**' operator we can assign the elements to variables, for use later on in the code.

8.1.8 Assignments using the '+++' operator

We have just seen the use of the '++' operator to assign variables. The '+++' operator is similar to '++'. It also works with String, Bytes and Sequence types. But while '++' allows matching the left most subsequence, '+++' allows matching right most subsequences .

For example, given :

```
var paragraph : String = 'First sentence. Second sentence. Last
sentence.'
```

with the filter :

```
V # ?v1 ++  %'.'++ ?v2
```

…we get : `v1 == 'First sentence'` and `v2=='` Second sentence. Last sentence.`'`.

with the filter :

```
V # ?v1 +++  %'.' ++ ?v2
```

…we get : v1 == ' First sentence. Second sentence' and v2==' Last sentence.'.

8.1.9 "and" in filters, "or" in filters

The boolean combinators "and" and "or" can be used in filters as shown in the following examples.

In the example below, we test that aString includes both 'dog' and 'cat' as substring (in any order).

```
aString  #  (? ++ %'dog' ++?) and (? ++ %'cat' ++?)
```

In next example below, we check that aString begin with 'One upon a time' and assign the content of aString to the variable 'story' .

```
aString  #  ( %'Once upon a time' ++?) and (?story)
```

In the example below, we test that aString includes either 'dog' or 'cat' as substring (in any order).

```
aString  #  (? ++ %'dog' ++?) or (? ++ %'cat' ++?)
```

In the example below, we check that aString begin with 'One upon a time'. If it does not, we assign the content of aString to the variable 'notAStory' .

```
aString  #  ( %'Once upon a time' ++?) or (?notAStory)
```

8.1.10 Testing a multiset

We saw previously that if a multiset == {'Circus-DTE'}, then the statement below will evaluate to true:

```
cMultiset # {%'Circus-DTE'}
```

However, if the multiset == {'Circus-DTE','problem'} then this rule is false. Why? Because the rule is checking for only one occurrence of any element within the multiset, we need to add the following:

```
cMultiset # {%'Circus-DTE'} ++ ?
```

This now will search a multiset for any element equalling 'Circus-DTE'.

8.1.11 Testing a sequence

```
cSequence # ?++ %[2,3]
```

This example, above, tests whether a given Sequence of Ints finishes the sequence with the elements [2,3] – this is possible because sequences have set order.

8.1.12 Testing a tuple

```
cTuple #   <%'A simple Tuple',?>
```

The example above tests to see whether cTuple contains 'A simple Tuple' as it's first parameter, along with a second parameter which is tested for existence – but no more. For example this rule will match the following tuples:

`<'A simple Tuple', 'xxx'>, <'A simple Tuple',9.87>`… etc

This rule, however will not match a tuple such as:

```
<'A simple Tuple','xxx','an extra element'>
```

8.1.13 Testing a dictionary

```
cDico # ? ++ {'Anna' = ?age}
```

The example above tests if there is an element with key 'Anna' in the dictionary and assign the variable 'age' with the corresponding value. This filter will for example match the dictionary :

{'Victor': 10, 'Lewis': 12, 'Anna' : 7, 'Barbara' : 4, 'Leon' : 9}

and the variable 'age' will be assign the integer '7'.

8.1.14 Testing a structure

```
// One can write :

cStructure #   ?++ <age=?theAge > ++?

// Or :

cStructure #   <[age=?theAge ]>
```

The example above tests if there is a field 'age' in the sequence and assign the variable 'theAge' with the corresponding value. This filter will for example match the structure :

<noun='Anna', age=7, nationality='German'>

and the variable 'theAge' will be assign the integer '7'.

Note that the following filter is not equivalent :

```
cStructure #   <age=?theAge >
```

Indeed, this last filters only matches a structure with exactly one field named 'age'.
So, in particular, it does not match the structure <noun='Anna', age=7, nationality='German'> defined above.

8.1.15 Testing a referenced object

Syntax is as follows :

```
<subject> # ref <filter expression>
<subject> # wref <filter expression>
```

The keywords ‘ref’ or ‘wref’ are used to express de-referencing in filters. They are dual of the ‘!’ operator used for de-referencing through imperative expressions.

```
x # (?:ref String and ?aRef)
```

In the example just below, the filter matches if the subject *x* has type `ref Any` and in that case the variable *aRef* will have type `ref Any`. If *x* has type `wref <Any,Any>`, the filter also matches because `wref` is a sub-type of `ref` but the variable *aRef* will still have type `ref Any`. This is a mean to manipulate writing references like reading references, making sure that one does not modify the referenced object.

```
x # (ref <?,?> and ?aRef )
```

In the example just below, the filter only matches if the subject *x* has type `wref Any` and the variable *aRef* will have type `wref Any`. If *x* has type `ref Any`, the filter won’t match.

```
x # (wref ? and ?aRef )
```

8.2 The 'Rule'

The above examples of pattern matching are only half-complete. They are the left-hand side of rules, which if evaluated to true, will execute the right-hand side section. This part, the right hand side, is currently missing from the examples above. To separate the left-hand side rule, from the right-hand side, Circus-DTE uses the '=>' symbol.

A rule can be expressed as:

Subject # Filter => Statement to compute if Pattern Match is successful.

In other words, a rule is the equivalent of an '*if then*' statement, but with additional features (mainly, scope introduction).

So, to code the following:

"*Print the remaining elements from the Sequence of Ints that start with the pattern [1,2]"*

The lambda function, below, could be implemented:

```
const testSeq:=lambda seq:[Int*] .( seq # %[1,2] ++?rem => print(rem))

const cSequence:=[1,2,3,4,5,6]
const test:=testSeq(cSequence)
```

As expected, the result is:

```
test.ci ...
```

```
[3,4,5,6]
```

8.2.1 Simple Rules

As we have seen previously, it is not permitted within Circus-DTE to implement an "if/then" code block with no "else". We must express this as a rule. Rules do not necessarily have to include a complex filter and expression, they may simply be made up of simple operator based expressions.

For example:

```
(A < 0) => print("A is less than zero!")
```

8.2.2 Multiple rules

Rules may follow other rules just like nested if/then statements in most high level languages. Each rule, if nested within the preceding rule's brackets will be executed, if this former rule was evaluated to True.

For example:

```
(a>b) => (a>c) => (a==d) => <some function...>
```

Here we see that there are 3 cascading rules, where the last two are only executed if the rule preceding it is evaluated to True.

8.2.3 Extraction using rules

In a **Multiset**, *x*, containing element *e*, the following pattern matching operation will remove element *e*:

```
x # {%e} ++ ?y => x:=y
```

The rule simply takes the remaining elements from the Multiset and reassigns them to x (via a temporary variable y, should the element e appear within x), effectively overwriting the original Multiset with the new variable.

It should be noted that if *x* contains one or more elements *e* (as a Multiset may do), only one element will be removed. Because there is no defining order within a Multiset, trying to predict which element will be removed, (providing it is equivalent to *e*), is not possible.

We are able to expand the filter by adding further elements. For example:

```
x # {%e,?,%7} ++ ?y => x:=y
```

This would remove 3 elements (assuming there were at least 3 elements in the multiset) so if we assume the multiset contained the following elements:

{1,2,3,4,5,6,7,8,9,10} and if e==10

We would be left with: {1,2,3,4,5,6,8} or perhaps:
{1,2,3,4,5,6,9} or
{2,3,4,5,6,8,9}… etc

(i.e. we can not predict which element will be removed from the '?' part of the filter.)

Finally we can include the variable assignment operator within this filter, for example:

```
x # {%e,?t,%7} ++ ?y => x:=y
```

Here we see that assuming:

var t is declared as an Int

Following the successful rule evaluation, t would be assigned to one (random) element, within the multiset.

8.2.4 Using dictionaries and rules

Dictionaries are provided as a basic type within Circus-DTE to facilitate the access/lookup functionality, which they have been designed around. The example below shows how to assign a name, from an id value (the unique key) with the use of a rule, taking a particular dictionary element from within the known list of ids:

```
module dictLookup

{
  type nameIDPair={String:String}
  const myIds:nameIDPair={"1a"="John", "2b"="Trevor","3b"="Mark"}

  const nameFromID:=pam id:<String,nameIDPair>,out:String
  .(
     id # <?idToFind,?idMultiSet>    =>
     idMultiSet # {idToFind=?out}++? => unit
   )

  // Test with
  const t:=nameFromID(<"2b",myIds>)  verify (t=="Trevor")

}
```

8.2.4.1 Dictionary lookup explanation

First we create our nameIDPair type which is a basic unique key string, paired with a value of another string.

Then we assign a const to equal our contact addresses. The variable 'myIds' now holds 3 such pairs, and is of type nameIDPair.

We now define a pam which accepts a tuple; the id we are looking for, and the dictionary in which it's stored in. We define the pam to return the String, out, which will contain the name of the contact found.

Following this, 2 variables are defined in the first filter, which serves to gain access to the two individual variables for use in the next rule. This rule discovers the value of the dictionary entry, indexed by the key, *idToFind*.

Notice how the rule for dictionary lookups is formed with:

```
<unique key> = ?<var>  …
```

Here we assign the value of the name found to the variable (in the above case: *out*) and complete the rule construction with '++ ?'. This part ('++ ?') is important, because without it, as with multisets, we would only match this pattern, should it be the only element within the dictionary. Obviously as we have multiple elements in our dictionary, this is not the desired behaviour.

Although we do not directly use the *found* variable in the above example, it is included for completeness. In a proper program you may wish to decide what to do, should the key not be included in the supplied dictionary (in our case, *out* will remain at its default value, i.e., '')

8.3 Exercises

1) Write a pam which when passed a multiset of integers or floats, returns a tuple which contains the smallest value in the variable and also the remaining elements of this multiset.

8.4 Summary

Pattern matching is made up of a Filter, a Pattern and then the Rule to execute, should the expression of the match between the pattern and the filter evaluate to successfully.

The Rule operation is denoted by a '=>' symbol.

'%' *contains the following item*". This can be a variable or a further expression.
'…++?' *the zero or more remaining elements after the pattern*
'?++…' *the zero or more remaining elements before the pattern*
'++?v','?v++' *assignment of the additional elements to variable v*
'?v' *assign ythe subject to v; if the variable exists, may perform a run time type checking – see later*
'?v:t' *assign to the variable v, if the subject has a compatible type (may be checked at run-time)*

Rules can be broken into subparts:

	If = = true	***rule***	then do…
	subject # filter	=>	action
or more simply:	(boolean expression)	=>	action

Step 1: Evaluate the filter,
Step 2: Perform the pattern-matching test
Step 3: If it does, perform the right hand side, in the scope defined by the filter (variable introduction).

9. Iterations

The symbol '*' (asterisk) is used to declare an iteration. The iteration of the loop is only stopped when the actual code block within the loop evaluates to **none**. This would usually occur in the form of a rule that is nested at the top level in the loop.

To demonstrate a simple iteration, looping through a Multiset of Strings, the following example is used:

```
const testPAM:=pam x:{String},output:None .(
        *(
            x # {?a}  ++ ?y => (
                print(a);
                x:=y
                )
         )
       )

const t:=testPAM({"this","is","a","test","of","iterations"})
```

The code is simply printing all values of a multiset (no order is preserved!).

9.1 Breaking an iteration loop

To halt the operation of an iteration loop, the code within must either evaluate to **none**. This can be generated with a rule, as we can see above.

This code demonstrates that the loop encased between iteration construct '*(…)', is iterated through until the rule is evaluated to be **none**. Once this happens the loop is exited.

The code produces the following output:

```
'test'
'iterations'
'this'
'is'
'of'
'a'
```

We see the result demonstrates nicely, that Multisets have no predefined order.

This code shows how it would be possible to extract all the square numbers from a Multiset of Ints, using a rule within an iteration:

```
const testSquares:=lambda myMS:{Int}
         var a:Int=1
         var c:Int=1
         var result:{Int}.(
         *(
           myMS # {%a}++? => (
              result := result + {a};
              c := c+ 1;
              a:=c*c
          );
         print(result)
         )

const cMulti:={1,2,16,4,5,6,8,27,9}
const test:=testSquares(cMulti)
```

This demonstrates the rule that says that to continue the iteration, we must find a square number within the Multiset. Upon first look, it seems as if the '++?' is doing nothing in this rule, as it is explicitly saying that the Multiset must contain a further zero or more elements. It actually is ensuring we do not just specify that the Multiset should be a single 'square number' (which is what the rule 'mySeq#[a]=>….' would imply)

The output of the above program:

```
{1,16,4,9}
```

A more advanced version of the above program could be :

```
const testSquares:=pam myMS:{Int}, result:{Int}
         var c:Int=1.(
         *(  myMS # { %(c*c) and ?a}++? => (
              result += {a};
              c += 1
              )
           )
         )
verify (testSquares({1,2,16,4,5,6,8,27,9})=={1,4,9,16})
```

9.2 Order is important

The innermost loop (see below) of any iteration block has important order considerations. If the **print**(a) statement followed the `x:=y` line then this loop would evaluate to ***none***, because this is what the **print**(a) is evaluated to. This would have the effect of stopping the loop after one iteration.

```
print(a);                   // this evaluates to none
x := y                      // this evaluates to unit (see below)
```

To avoid this, we either ensure that the imperative statement **x:=y** appears last, or alternatively, and generally more safe, we could turn the **print**(a); command into `n:=print(a)` [where n was of type **Bool**]. Finally we could use the **unit** keyword, as the final expression, within the code block to force this block into evaluating to **unit**.

9.3 Creating Iterating Pams

We also note that this pam is effectively completely encased within a loop. This, for the function we are writing, is not a problem - however, there are many pams where the iterative nature would be an additional feature which we would have to create an extra pam for.

However, it is possible to call the pam in an *iterative* mode, thus eliminating the need to write another pam. If we first remove the iteration loop from the main code and call the pam like so:

```
const testPAM:=pam inp:{Any},output:None
        var a:String.(
           inp #{?a}  ++ ?inp => printS(a)
        )

const t:=*(testPAM) ({"this","is","a","test","of",6,"iterations"})
```

We create the desired effect, giving the iteration command to a pam, rather than the pam iterating itself. This actually creates at compilation time a new, iterating pam.

9.4 Exercises

1) Write a pam which when passed a string and a character (a one length string) returns the number of times this character appears within the String.

9.5 Summary

Iterations are defined using the '*' operator with the code or pam that is to be iterated encased within parenthesis.

It is possible to create iterating pams by encasing them within the iteration construct and parentheses.

Iteration blocks are executed until the block evaluates to **none**, which maybe caused by a rule not passing the execution or perhaps the result of a pam, encased within the iteration loop.

10. Recursion

Circus-DTE also allows definition of recursive pams and lambdas. Such pams and lambdas require explicit type definition.

```
//pam to check that all elements in a polymorphic sequence are the same
const allElemIdentical : [Any*] ==> Bool = pam x:[Any*], y:Bool.(
[|
  (x == []) => y := true,
  x # [?elem] ++ ?rest =>
   [|
     rest # %[] => y := true,
     rest # [%elem] ++ ? => y:= allElemIdentical(rest),
     y := false
   |]
|]
)
```

Cross recursion in pam and lambda definitions is allowed. For example you can define a pam P that calls a pam P' which itself calls P.

You can of course combine iteration and recursion in pam and lambda definitions.

```
// type tree of strings
type Tree = <node : String> | <node : String, sub :[Tree*]>

// pam to linearize a tree of strings
const linearize :[Tree*] ==> String =
pam tree :[Tree*], y : String .(
*( tree # [?x]++?tree  =>
   [|
   x # <node=?label,sub=?seq> =>  y += ' '+label+' ' + linearise(seq),
   x # <node=?label>          =>  y += ' ' + label
   |]
))
verify (linearize([<node='beginning',
                    sub=<node= 'middle',
                         sub = ['end']>>])
      ==
      'beginning middle end')
```

10.1 Exercises

TBD

10.2 Summary

You can use recursion and cross recursion when defining pams and lambdas.

11. Testing types at run-time

As has been previously discussed, it is useful for pams to handle different **type**s and to perform different actions, dependent on the type of object received.

In this example we assume we have two defined types: 'lorry' and 'car'. We would like a pam to accept both types and depending on what type is passed to the pam, we perform a different action on each.

In this example we decide to paint lorries *blue* and cars *red*, using one painter pam.

First our Car structure, defined as follows:

```
type Car = <colour:String, seats:Int, engineSize:Float>
```

Here is our Lorry structure – notice that it is similar, but not identical to the Car structure. This will cause us problems, as we will see below:

```
type Lorry = <colour:String, storageCapacity:Int, engineSize:Float>
```

We now write our pam definition, which will accept either a lorry or a car. It will return a modified variable, containing the newly coloured vehicle.

```
const painter:pam= vIn: Car | Lorry,vOut: Car | Lorry.(

)
```

Here we see our first problem: The Circus-DTE compiler will create a default value for vOut. This will be an undetermined type (either Car or Lorry), but we don't know which one. By the time we access vOut, it could already be either of type Lorry or Car. We already know that vOut is a type which is a Union between a Car and a Lorry. By assigning it directly from vIn at compilation time, we are changing the basic type to that of vIn. To prevent this happening we must execute this assignment at run time, which requires the use of temporary variables, in this case one for a Lorry and one for a Car and a rule.

```
      var aCar:Car
      var aLorry:Lorry.( <snip> )
```

Now to perform the type checking we follow this with:

```
      vIn # ?aCar   =>  ( <snip> )
      vIn # ?aLorry =>  ( <snip> )
```

The code that has been removed is the code that should be run should the preceding rule be successful. The rule is seeing whether our temporary variables "fit" the input, vIn, and in doing so, essentially check for *type similarity*. If the variable vIn is able to be assigned to aCar then we know that vIn, (and obviously the compiled-type aCar), is of type **Car**.

By the time we enter our code which executes on success of the rule, we are able to use either aCar or aLorry (depending on which rule was successful), which will be a placeholder for the original variable.

```
        vIn # ?aCar =>(
                vOut:=aCar;
                vOut.colour:="red");

        vIn # ?aLorry =>(
                vOut:=aLorry;
                vOut.colour:="blue"
                )
```

Notice that the assignment vOut:= … will be done at run-time, as opposed to compile time as it is encased within a rule. This ensures we are safely assigning vOut, to the correct type.

Finally the testing code:

```
const myLorry:Lorry=
                   <colour="black",storageCapacity=1000,engineSize=5.4>
const myCar:Car=
                   <colour="black",seats=4,engineSize=1.6>

const paintedLorry := painter(myLorry) verify(print(paintedLorry)==none)

const paintedCar   := painter(myCar)   verify(print(paintedCar)==none)
```

…and the result, after initialising both vehicles to the colour "black", then *painting* each:

```
<engineSize=5.4,colour='blue',storageCapacity=1000>
<seats=4,engineSize=1.6,colour='red'>
```

11.1 Action Systems

We can also use this example to demonstrate the use of the '[|' and '|]' constructs. The use of these symbols effectively encases the complete block of code.

The '[|' followed by '|]' specifies a block of code to be performed as a controlled sequence of events, each separated by a comma. The comma is separating each rule and will act as a "*switch*", so if the first rule is executed (because the pattern match evaluated to **true**) then the remaining items, after the comma will not be tested. This continues until either the end of the block has been reached, or one expression has been evaluated to **unit**.

The separator of each block may be a '**Then'** which signifies the following expression should be evaluated, only if the expression was evaluated as True, or successfully or an "**Else**" separator which only evaluates the following expression only if the preceding one evaluated to **none**.

Although it makes little difference to our example, the following code segment is more accurate than our previous examples:

```
        [|
          vIn # ?aCar =>
           (
                vOut:=aCar;
                vOut.colour:="red"
           );
        ,
          vIn # ?aLorry =>
           (
                vOut:=aLorry;
                vOut.colour:="blue"
             )
        |]
```

Of course it is possible to combine multiple separators within the different blocks, so it is perfectly possible to have:

```
       [|
           vIn # ?aCar =>
            (
                  vOut:=aCar;
                  vOut.colour:="red"
            );
       ,
           vIn # ?aLorry =>
            (
                  vOut:=aLorry;
                  vOut.colour:="blue"
            )
       |]
Else
       [|
             //not a car or a Lorry!
             Errorhandler()
       |]

       //…etc
```

11.2 Summary

Type checking at run time must be achieved with a combination of rules and temporary values, one rule and temporary value per type you wish to check. We must conduct type checking like this, at run time, because the compiler will attempt to *type* variables, which in the case of Union types, is unpredictable.

The use of the [| and |] encasing a section of code enables us to put a form of control into the ordering and execution of expressions. These orders maybe combined to produce complex order control, with the use of many separators in succession (a **Then** (b;c;d) **Else** e **Then** f , *etc*), where a, b, c etc are blocks of code encased by '[|' and '|]'.

12. Further Composition of Pams

13. Concurrency

Concurrent programming allows a programmer to access and store shared variables in a common "pool" of memory – called the "Co-ordination Memory". This coordination memory (*CM*) can be thought of as a globally available Multiset, with synchronisation properties.

There are 5 commands used in concurrent programming, these are described below:

13.1 The put command

Usage: ***put*** *<expression>*

The put command places an expression into the co-ordination memory. This command, with just an expression, places the expression into the CM typed as *Any.* If the type information is included with the expression this information will be used instead.

13.2 The read command

Usage: ***read*** *filter => <expression>*

The read command uses a ***filter*** (see the section on Pattern Matching for more details on *filters*) to retrieve an expression from the CM. Once obtained, this expression remains in the CM, and the read function will return **true**. The function 'read' is executed without blocking – i.e. a wait state is not invoked if no expression is found – the function will simply return **false**.

13.3 The bread command

Usage: ***bread*** *filter => expression*

'bread', the blocking read command, uses a ***filter*** (see the section on Pattern Matching for more details on *filters*) to retrieve an expression from the CM. Once obtained, this expression remains in the CM. The function 'bread' is executed with blocking – i.e. a wait state is invoked until an expression is found. Therefore we can see that 'bread' always returns **true**.

13.4 The get command

Usage: ***get*** *filter => <expression>*

The get command uses a ***filter*** (see the section on Pattern Matching for more details on *filters*) to retrieve an expression from the CM. Once obtained, this expression is removed from the CM, and the 'get' function will return **true**. The function 'get' is executed without blocking – i.e. a wait state is not invoked if no expression is found – the function will simply return **false**.

13.5 The bget command

Usage: ***bget*** *filter => <expression>*

'bget', the blocking get command, uses a ***filter*** (see the section on Pattern Matching for more details on *filters*) to retrieve an expression from the CM. Once obtained, this expression is removed from the CM.

‘bget’ is executed with blocking – i.e. a wait state is invoked until an expression is found. Therefore we can see that ‘bget’ always returns **true**.

13.6 Dining philosophers

To see the co-ordination memory in practice, the following well-known concurrency problem ‘Dining Philosophers’ is used to demonstrate the usage of concurrent programming.

The story takes place in a remote monastery where five monks are devoted exclusively to philosophy. They would gladly spend all of their time thinking, but they also have to eat from time to time. At the centre of their shared table is located a spaghetti dish which is supposed to be always filled. There are five plates and five forks. Two forks are required to eat spaghetti. Any philosopher who wants to eat stops thinking, eats, and then goes back to its thoughts.

The problem here is to find a protocol enabling the philosophers to have a little snack.

```
module Philo{

const eat:=pam x:Int,y:Int
.(
 // send the philosopher to eat for a random amount of time
 // (between 0 and 5 seconds)

   printS(str(x)+’ starts eating’);
   printS(str(x)+’ stops eating’);
   y:=int(round(rand()*2.0))
)

const NbPhilo:=5
const modNb := mod(NbPhilo)

const phil := lambda lifeDuration:Int
      lambda p:Int
         pam x:Void,y:None
            var fl:String=’fork’+str(p)
            var fr:String=’fork’+str(modNb(p+1))
            var dc:Int=lifeDuration
      .(
            *(
                  (dc>0)
                   => bget %fl
                     => [|
                         get %fr
                         => (
                             dc:=dc-eat(p);
                             put fr;
                             put fl
                            )
                         ,
                          put fl
                        |]
              );
      print(str(p)+’ dies’)
      )


const LifeDuration:=10
```

```
const phil0:=phil(LifeDuration)(0)
const phil1:=phil(LifeDuration)(1)
const phil2:=phil(LifeDuration)(2)
const phil3:=phil(LifeDuration)(3)
const phil4:=phil(LifeDuration)(4)

const main:=pam i:[String],o:Int
.(
    put 'fork0';
    put 'fork1';
    put 'fork2';
    put 'fork3';
    put 'fork4';
    (phil0() || phil1() ||  phil2() || phil3() || phil4() )

 )

}
```

13.6.1 Brief Explanation

Using the blocking get functions we are able to allow our philosophers to ability to attempt to get the left fork (*fl*) and when this is successful (when, rather than if, given we are using a blocking get) we remove this left fork and attempt to take a right fork.

If this is successful we allow the philosopher to eat, and then return both forks, otherwise we return the already-obtained left fork and wait for a random time, to try again.

We can see from the main pam that we in fact begin our concurrent programming model by using the “during” pam composition mode (‘||’).

13.7 Summary

Concurrent Programming allows the programmer to gain access to a memory area called the “co-ordination memory”.

There are 5 commands used to insert, remove or read expressions from this memory, they are put, get, bget, read and bread.

The read commands do not delete the expressions that they find in the memory, whilst the get commands remove expressions.

Blocking commands wait until the filter is matched before returning from within the function.

14. The use of Modules

As we have seen earlier all Circus-DTE code is identified within a module. A module is a useful container for functions, etc, which fall into a similar category, or which are written to perform one larger action. For example, it is feasible to use a "Maths" module, or a "Sets" module. This ensures that when looking for prewritten functions, other people have a chance to access all relevant code, in one file.

Module names are usually started with a capital letter and may not begin with a numerical value.

There are no limitations on the number of modules contained within one Circus-DTE program. The code below, held in one file, is perfectly valid:

```
module Mod1{

 const cm1: Int = 7

}

module Mod2{

 const cm1: Int = 20

}
```

Notice that it is valid to also use the same variable names across different modules.

On compilation these modules are stored in Circus-DTE compiled code with an underscore prepended to the module name (with a .pyo extension). For example, when the above file is compiled the following two files are created:

```
-rw-r--r--   1 ciruser  xerox         85 Jun  27 19:56 _Mod1.pyo
-rw-r--r--   1 ciruser  xerox         88 Jun  27 19:56 _Mod2.pyo
```

14.1 The Import command

It is possible to import any **lambda function**, **pam**, **const** or **type** into a new module with the "**import**" command.

The basic syntax is as follows:

```
     from <module name> import <object to import>
```

So, to demonstrate this, to access the **cm1** constant from **Mod1** we would simply write:

```
module ImportTest{

 from Mod1 import cm1
 const c2 := cm1 + 10 verify (c2==17)

}
```

Of course, now if we also wish to import **cm1** from **Mod2**, we have a problem, as the two variables have a naming conflict.

14.1.1 Renaming imported definitions: The '*as*' keyword

Extending the **import** command we see that it is possible to import and rename a function at the same time. We are given this opportunity so that we can resolve naming conflicts.

```
from <module name> import <object> as <new object name>
```

Below is an example of how to import both **cm1**s (from the above modules) and rename them in order to specify which one to use.

```
module ImportTest2{

 from Mod1 import cm1 as cm1a
 from Mod2 import cm1 as cm1b

 const totcm := cm1a + cm1b (verify totcm==27)

}
```

14.1.2 Importing all definitions: The '*' keyword

All definitions of a given module M can be imported through the "**from M import ***" statement. The compiler detects and reports potential name conflicts.

```
module ImportTest3{

 from Mod1 import *

 const totcm := cm1 + cm2 verify (totcm==27)

}
```

14.2 Summary

Multiple Modules may be contained in one Circus-DTE file. Each module however should contain related functions and code, to ease the use by other programmers who may be in a position to share this code, in the future.

Each module once compiled, is stored within a file on the local file system which is named "_" + the module name, with an extension: ".pyo"

Lambda functions, pams, consts and types may all be imported and renamed, if necessary, by the "**from**… **import** … **as**" command. All definitions of a given module can be imported thanks to the "**from M import ***" statement.

15. Glossary

A list of terms used through out this tutorial:

Boolean: A type containing three values: true, false and none.
Boolean operators: operators that are used to derived a boolean result. For example: ’==’, ’>’, ’>=’, ’<=’, etc.
(Complex) Expression: a statement which may encapsulate a function, calculation or operation.
Constant: A defined variable, which may not be modified after definition. Constants, declared with **’const’**, may be lambda functions, pams or any value.
Dictionary: A type that specifies a unique key and a value, which is associated with that key.
Enumeration: A collection of values, used to make a user defined type.
Filter: may be defined as expression to check a given pattern against. Filters are commonly variables, but may include operators such as concatenation or return values from functions.
Float: a type used to represent floating point numbers – defined as IEEE double.
Function: a procedure, discrete contained code, which may take a parameter and may return a value. **Pams** and **Lambdas** are both "Functions"
Imperative Expression: Statements such as assignments, rules, iterations, or a sequence of expressions.
Int: a type which is used to represent whole numbers. The size of which is unbounded.
Iteration: A loop, constantly evaluated until the encased block computes to ***none***.
Lambda: a type of function which takes a typed input value and operates a set of core functionality, based on this value.
Module: a set of correlated constants and functions, delimited with braces - '{','}' and named accordingly via the Module command.
Multiset: A built in type, which stores variables of another specified type, in no order, with duplicates allowed.
Pam: a function which takes a names input and output type, and performs a block of code, in order to modify the output variable type.
Rule: Defined with '=>' which has a left hand side which is evaluated and a right hand side which is executed, should the left hand side expression evaluate to **true**.
String: a Circus-DTE type which represents characters, in a particular order.
Tuple: A named set of a collection of types, in a sequence, which may be accessed by the index value of the position in the list.
Type: a named entity, which represents the values which variables may take. For example the **Bool** type dictates that the legal values are **true**, **false** and **none**.
unit: a return value used by the internal computation of Circus-DTE. Assignments return **unit**, which is a value *(the only one*) of type **Unit**.
Unit: The Unit type with one value, **unit**.
Value: a legal representation, stored in a variable, of a given object type
Variable: a named object that holds a value of a given type declared with the keyword '**var’**. May only be defined within **pams** or **lambda functions**.
Void: A type that is used during the definition of a pam or lambda function, to indicate that no parameter is required.

16. List of Circus-DTE Keywords

and
as
Before
Bool
const
do
during
else
Else
false
Float
for
from
if
import
in
Int
lambda
module
none
None
or
pam
then
String
Then
to
true
True
type
unit
var
verify
Void

16.1 List of Circus-DTE Operators/Symbols

< less than, inclusion
> greater than, inclusion
<= less than or equal to, strict inclusion
>= greater than or equal to, strict inclusion
== equivalency check
!= not equal to
:= assignment
? Pattern Matching test for existence (with a variable to achieve run time check and assignment)
++ Pattern Matching test (and possible assignment) for elements preceding the match item
+ addition/concatenation
- subtraction
/ division
[.. : ..] slicing
* multiplication or iterating construct
.(…) new scope (follows variable declarations)
; imperative statement separator
{..} module container, multiset constructor
(…) brackets used to denote ordering/grouping.
=> rule operator
filter operator
% pattern identifier
[…] sequence container, index operator
<…> tuple, structure and dictionary constructor
{..=..} dictionary key/value binding
[|…|] symbols used to encase blocks of sequenced code
, general separator
" and ' string delimiters
\ literal character identifier
|| during operator, for pam composition.

17. Index

- ‘++? *var*’ and ‘? *var* ++ ’ operators, 48
- ‘+++’ operator, 48
- ‘<’ and ‘<=’ Operators, 41
- *as*, 67
- bget command, 63
- Booleans
  - 3-State Logic, 15
- bread command, 63
- Bytes
  - type, 18
- Casting Variables, 33
- choice function, 32
- chr Function, 31
- Circus-DTE
  - Requirements, 8
  - version, 8
- Compilation, 12
- Compiler
  - options, 12
- Compiling, 10
- Concurrency, 63
- **constant** declaration, 11
- constructed type
  - structure, 20
- coordination memory, 63
- Default Values, 33
- Dictionaries, 22
  - addition, 22
  - items, 22
  - keys, 22
  - look up functions, 53
  - subtraction, 22
  - values, 22
- Dining philosophers, 64
- **Else**, 60
- Enumerations
  - example of, 27
- equivalency check, 14
- Filters, 45
- float function, 31
- **Float, Floating Point Numbers**, 16
- for .. do loop, 39
- `from`, 67
- get command, 63
- If, then, else, 40
- Import, 66
- import *, 67
- `in command`, 26
- index, 17
- **Ints, Integers**, 16
- Iteration Constructs, 55
- Lambda Functions, 35
  - Anonymous declaration, 37
  - calling, 35
  - defining, 35
  - example of, 36
  - `implies`, 37
  - `multiply`, 37
  - usage, 35
- Modules, 66
- Multisets, 18
  - addition, 18
  - division, 19
  - multiplication, 19
  - Order, 18
  - subtraction, 18
- None type, 23
- Numbers, 16
- ord function, 31
- Pams. *See* Polymorphic Abstract Machines
- Pattern Matching
  - assignment with a pattern, 46
  - declaring variables inside filters, 46
  - matching with a pattern, 45
  - Testing a multiset, 49
  - Testing a sequence, 49
  - Testing a Tuple, 49, 50
- Pattern-Matching, 45
- Polymorphic Abstract Machines, 42
  - composing, 43
  - composition of, 62
  - definition, 42
  - iteration of, 56
  - passing parameters by copy, 43
  - usage, 42
- put command, 63
- rand function, 32
- Randomising Functions, 32
- Range Function, 23
- read command, 63
- Records, 21
- round function, 31
- Rules, 51
  - expression of, 51
  - extraction, 52
  - multiple, 52
  - using dictionaries, 53
- Sequences, 19
  - concatenation, 20
  - division, 20
  - example of, using rules, 51
  - multiplication, 21

slicing, 21
subtraction, 20
Simple type
Bytes, 18
Slicing, 17
str function, 31
Strings, 16
comparison, 16, 18
division, 17
multiplication, 17
Structure
type, 20
Structures, 21
Sub Types, 25
Subtraction, 17
Then, 60
Tuples, 21
type
Bytes, 18
Structure, 20
Types
declaring, 15, 26
testing at run time, 58
union, 26
Unit Type, 24, 25
var keyword, 29
Variables, 29
scope of, 30
Verbose output, 12
verify command, 14
Void type, 23

18. References

Circus-DTE JVM Platform Design: **Philippe Rérole**, XRCE. May 1999
Circus-DTE: an overview and some key points: **Jean-Yves Vion-Dury**, April 16, 1998

19. Acknowledgements

Jean-Yves Vion-Dury, Philippe Rérole, Laurence Hubert, Gilbert Harrus, Veronika Lux, Thierry Jacquin, Emmanuel Pietriga, Michel Gastaldo

20. Appendix A: Solutions to the exercises

20.1 Circus-DTE Types

1) Define a Type, *EvenIntsLTTen*, that may only contain even Integers, which are less than 10, which are stored in no particular order.

```
type EvenIntsLTTen = Int in {2,4,6,8,10}
```

2) Define a simple date structure, suitable for containing dates in the form dd/mm/yyyy

```
type date = <date:Int in {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,
18,19,20,21,22,23,24,25,26,27,28,29,30,31}, month:Int in
{1,2,3,4,5,6,7,8,9,10,11,12},year:Int>
```

3) Define a Type, *Contact*, based on a structure that represents a contact found in a typical address book, able to store names, telephone numbers and an address.

```
type Contact = <name: String, birthday: date, telephone: String,
address: String>
```

4) Define a Type, *ContactWithFullName*, which contains this structure, above, with a unique full name as a pair – with the principal *concept* of being able to retrieve a person's telephone number by their full name.

For this we should use a *dictionary* type, and reference each contact with a unique String, in this case their name. Note, if two or more contacts had the same name, the names must be modified in order to reflect this (by including a middle name, etc)

```
type ContactWithFullName={String:Contact}
```

20.2 Circus-DTE Variables

1) Write a simple Circus-DTE program that proves that the length of an empty string is equal to the value of a float variable, set to its default value.

```
module myCircus-DTETest{

 const myString : String=""
 const myFloat := 0.0 verify (card(myString)==(int(myFloat)))
}
```

2) Create a constant, *NameList*, which holds a sequence of names (Strings) and have the program print out the first and last names, regardless of how many are in the sequence (assume at least one name is in the sequence)

```
module myCircus-DTETest{

 const NameList:[String*]=["Frank","Joe","Rio","Paulo"] verify
                           ((print((NameList[0:1])) == none) and
                            (print(NameList[-1,card(NameList)])==none))

}
```

3) Create 3 constants that represent possible values for the previous Contact type (see previous exercise) and then insert these into a new constant which stores them in the correct fashion, for the *ContactWithFullName* Type, also defined in the last exercise. *(see next page)*

```
module nameTest{

// from the last exercise

type date = <date:Int in {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16
17,18,19,20,21,22,23,24,25,26,27,28,29,30,31}, month:Int in
{1,2,3,4,5,6,7,8,9,10,11,12}, year:Int>

type Contact = <name: String, birthday:date, telephone: String, address:
String>


type ContactWithFullName={String:Contact}

// this question:

const contact1:Contact=<name="Paolo",
birthday=<date=2,month=5,year=1968>, telephone="555-6088",address="223
Weston Road">

const contact2:Contact=<name="Rio",
birthday=<date=12,month=3,year=1975>, telephone="555-6142",address="12
Hammerton Ave">

const contact3:Contact=<name="Harry",
birthday=<date=30,month=10,year=1954>, telephone="555-6551",address="77
United Terrace">

const phoneBook:ContactWithFullName =
                {"Paolo"=contact1,"Rio"=contact2,"Harry"=contact3}

}
```

4) Create a constant that is assigned a random name from the contact Lists constants defined earlier and print the phone number associated with this one from the same variable.

```
//assume above code (question 3) is within same module, or imported

const allContacts:{Contact}={contact1,contact2,contact3}

const randomContact:=choice(allContacts)
  (verify(print(randomContact)==none))
```

20.3 Lambda Functions

1) Write a lambda function that returns the square of any integer or float that is passed to it.

```
module lambdaTest{

  const squareIntOrFloat:=lambda src:(Int|Float)
   .(
    src*src
   )

  //test with

  const f:=4.0 verify (squareIntOrFloat(f)==16.0)
  const g:=3   verify (squareIntOrFloat(g)==9)

}
```

2) Write a lambda function which calculates and returns the answers to the following expression: (x*3)+(y*2)+(z).

```
module lambdaTest2{

  const exp:=lambda x:Int lambda y:Int lambda z:Int.((x*3)+(y*2)+(z))

  //test with…

  const a:=2
  const b:=3
  const c:=4 verify (exp(a)(b)(c)==16)

}
```

3) Without writing a new lambda function, but by extending the previous example, create a lambda function which calculates: 12+(y*2)+(z)

```
const exp:=lambda x:Int lambda y:Int lambda z:Int.((x*3)+(y*2)+(z))

// our new lambda function
const new:=exp(4)

const y:=3
const z:=6 verify (new(y)(z)==30)
```

20.4 Polymorphic Abstract Machines

1) Write a pam which when passed a multiset of integers or floats, returns a tuple which contains the smallest value in the variable and also the remaining elements of this multiset.

```
module pamMinTest{

  const multisetMin:=pam src:{Int},out:<{Int},Int>
    var minSoFar:Int
    var first:Bool=true
    .(
       for x in a do (
         //figure out lowest value
         (first) => (minSoFar:=x; first:=false);
         (x<minSoFar) => minSoFar:=x
         );

         //remove the lower value from the set, and set 'out'
         src# {%minSoFar} ++?newA => out:=<newA,minSoFar>

      )

  //test with…

  const f:={6,5,4,5,6} verify (multisetMin(f)==<{5,5,6,6},4>)
  const g:={6}         verify (multisetMin(g)==<{},6>)
  const h:={4,4}       verify (multisetMin(h)==<{4},4>)

}
```

20.5 Iterations

1) Write a pam which when passed a string and a character (a one length string) returns the number of times this character appears within the String.

```
module occurrences {

const occ:=pam src: <String,String>,out:Int
  var a:Int.(
   src # <?str,?chr> =>
        *(
           (a<card(str))=>( (str[a:a+1]==chr)=>out += 1; a += 1)
          )
   )

//test with…
verifyt (occ(<"abcdefgabaca","a">) == 4 )

}
```