A Gentle Introduction to Camelot

$Revision: 1.3 $

Hans-Wolfgang Loidl

**Institut für Informatik,
Ludwig-Maximilians-Universität München
hwloidl@tcs.ifi.lmu.de**

Kenneth MacKenzie

**Laboratory for the Foundations of Computer Science,
Edinburgh University
kxm@inf.ed.ac.uk**

This document gives an introduction to the resource-safe functional programming language Camelot. No previous knowledge of functional programming is assumed. *This is an early draft.*

* *I want to make this a document of learning by examples to show the main things of Camelot. It should not be a reference manual or anything like that.*

1. Introduction

While functional programming languages are great in providing a high level of abstraction, their resource consumption is fairly hard to predict. A lot of effort has gone into devising sophisticated compilation techniques to reduce the resource consumption. However, these techniques follow an all-or-nothing approach: either the optimisation will kick in, reducing resource consumption significantly, or it won't. In the latter case, the user usually has no way of identifying the reason, or to help the compiler in optimising the code.

Camelot takes the approach of giving the programmer more control about the space consumption of a program. In particular, it provides a structured way of gaining direct access to memory locations, so as to formulate code for in-place update. An extended type system is used to guarantee that the usage of these extensions does not break the referential transparency of the language.

Camelot is a language of the ML family. It is strict, first-order but allows for use-only higher-order functions. It includes object-oriented extensions, mainly for creating and using objects in an existing class hierarchy. Programmers familiar with ML or Ocaml should have no problems understanding the basic language promitives. In this introduction we will focus on the extensions for gaining direct access to memory, and how to use these for writing efficient code. Finally, we will discuss how the type system can ensure that the code is still safe.

This document aims to give an introduction to using the language and system without trying to be a comprehensive discussion of either. For more in-depth discussions of the language the reader is refered to papers on the language (TFP03,OCamelot) and the Camelot manual (Manual). In this document we first give a quick introduction how to compile and execute a simple Camelot program (Section 2). An example discusses the main characteristics of Camelot (Section 4). Then we discuss the language in more detail (Section 3). Finally, we give information on the system including profiling, tools etc (Section 6).

2. A Quick Introduction to Using Camelot

We assume that you want to compile the program `testList1.hs`which is included in the `examples` directory of the Camelot directory (available from our web Server (http://groups.inf.ed.ac.uk/mrg)).

2.1. Installing

You can download the Camelot compiler from the *Camelot Web Page* (http://groups.inf.ed.ac.uk/mrg/camelot/).

If you download a binary RPM, just issue the command

```
rpm -ivh <rpm-file>
```

If you download a source bundle, unpack it with

```
tar xvfz <tar-file>
```

Go into the directory `Camelot-distrib/src` and type **make**.

```
cd Camelot-distrib/src
make
```

See Section Section 2.4 about software you need to build the compiler.

Create a link to the top-level compiler and put it into your path, e.g.

```
ln -s camelot ~/bin
```

To test the compiler, go into `Camelot-distrib/examples` and follow the instructions in Section 2.2.

2.2. Compiling

Assuming the camelot compiler is in your path and that the Camelotlib.class is in your classpath, compile a Camelot program like this (the compile-time options are explained in *Compiler options*).

```
camelot testList1.cmlt
```

This will generate a Java class file `testList1.class`, which contains the JVM bytecode for the Camelot program. A second class file with the name `testList1$dia_0.class` will be generated, too. This file contains the bytecode for the memory management of the Java program, which depends on the data types use in your program and therefore has to be generated with every user program. You don't have to know any details about this class file, just be sure that both of these files are in your classpath when you execute the program.

2.3. Executing

To run the generated program just call the java interpreter, preferably JDK 1.4.xx (see Section 2.4, from the commandline (see the docuemntation on that Java interpreter on how to pass run-time options to VM), e.g.

```
java testList1 3 4 5
```

and you should see the following as output from this example program:

```
Input list:
 xs = [3, 4, 5]
Cloned list:
```

```
 ys = [3, 4, 5]
In-place update of first element in xs with 0 (nuke1st xs):
 zs = [0, 4, 5]
Cloned list:
 ys = [3, 4, 5]
Substituting 99 for 0 in zs (subst1 99 0 zs):
 zs' = [99, 4, 5]
Cloned list still is:
 ys = [3, 4, 5]
What a great system!
```

2.4. Software Dependencies

You should have the following software installed to compile and run Camelot programs:

- JDK 1.4.xx (to run the generated class file)

If you want to build the Camelot compiler from source you additionally need the following software:

- Moscow ML (to build the Camelot compile from source)
- OCaml (to build LFD module, only; OCaml page (http://caml.inria.fr/ocaml/))
- lp_solve (to build LFD module, only; download (ftp://ftp.es.ele.tue.nl/pub/lp_solve))

3. Camelot the Language

* *A more detailed introduction to Camelot. Should cover main language primitives, starting off with main concepts and linking them to existing languages. Focus on the diamond types*

3.1. Basic Camelot

The main characteristics of Camelot are:

Ocaml-like syntax

Call-by-value semantics

First order

Algebraic datatypes

Object-oriented extensions

Diamond types

3.1.1. Program Structure

A Camelot program consists of a (possibly empty) sequence of *datatype definitions*, followed by a sequence of *function definitions*. The function definitions can be preceded by an optional sequence of statements declaring the types of the functions, but we will make no use of this feature.

The language is designed for whole program compilation and therefore no module system is provided.

3.1.2. Types

Camelot has five primitive types: **unit**, **bool**, **int**, **float** and **string**. The unit type has a single value **()**, and the boolean type **bool** has two values: **true** and **false**. The numeric types **int** and **float** represent integers and floating-point numbers respectively; the values of both types occupy 32 bits. Finally, the **string** type has values which are (immutable) character strings. Camelot also provides array types and operations on arrays.

Here are some sample datatype definitions; note that datatypes can be both recursive and polymorphic.

```
type intlist = Nil | Cons of int * intlist
type 'a polylist = NIL | CONS of 'a * 'a polylist
type ('a, 'b) pair = Pair of 'a *'b
```

Datatype constructors must begin with an upper-case letter; all other identifiers begin with lower-case letters. Values belonging to such types are created by applying constructors in the usual way, and are deconstructed using the match statement:

* *Something on array types!*

3.1.3. Functions

Function definitions are introduced by the **let** keyword: as a simple example the factorial function could be defined by

```
let fac n = if n<0 then 1 else n*fac(n-1)
```

The match statement is used to deconstruct data types like this:

```
let length l = match l with
       Nil => 0
     | Cons (h,t) => 1+length t

let test () =
    let l = Cons(2, Cons(7,Nil))
    in length l
```

The form of the match statement is much more restricted than in SML or OCaml. There must be exactly one rule for each constructor in the associated datatype, and each rule binds the values contained in the constructor to variables (or discards them by using the pseudo-variable _. One may not use complex matches such as

```
match l with
   Cons(0,Nil) => 1  (* not valid in Camelot *)
 | Cons(1,Nil) => 3
 | _ => 0
```

To achieve effects such as this, one must use code such as

```
match l with
   Nil => 0
 | Cons(i,Nil) =>
       if i=0 then 1
          else if i=1 then 3
               else 0
```

> **Tip:** Note that constructor arguments are enclosed in parentheses and are separated by commas.

Function definitions and applications are written in a curried style:

```
let add a b = a+b
let f x y z = add x (add y z)
```

Note that Camelot does not support general higher-order functions. Therefore, any application of a function must involve exactly the same number of arguments as are specified in the definition of the function.

3.2. In-place Operations in Camelot

* *This section discusses, how to write Camelot programs that perform in-place heap operations. We use in-place list reversal as our main example.*

```
type ilist = !Nil | Cons of int * ilist
let rev l acc = match l with Nil => acc
                            | Cons(h,t) => rev t (Cons(h,acc))
```

In order to allow more precise control of heap usage, Camelot includes constructs allowing re-use of heap cells. Associated with a Camelot program there is a special JVM class known as the *diamond class*: the objects of this class are large enough to contain any instance of any datatype defined in the class. These objects are represented in Camelot as members of the special type ✧, and Camelot provides methods for manipulating the objects explicitly. This is achieved by equipping constructors and match rules with special annotations referring to diamond values. Here is the **reverse** function rewritten using diamonds so that it performs in-place reversal

```
let rev l acc = match l with
    Nil => acc
  | Cons (h,t)@d => rev t (Cons (h,acc)@d)

let reverse l = rev l Nil
```

The annotation ✧ on the first occurrence of **Cons** tells the compiler that the space used by the list cell is to be made available for re-use via the diamond value ✧. The annotation on the second occurrence of **Cons** specifies that the list cell **Cons(h,acc)** should be constructed in the diamond object referred to by ✧, and no new space should be allocated on the heap.

One may not always wish to re-use a diamond value immediately. This can sometimes cause difficulty since such diamonds might then have to be returned as part of a function result so that they can be recycled by other parts of the program. To avoid this Camelot provides a *freelist* for the storage of unused diamonds. This is accessed using another form of constructor annotation, which we will now describe. The alert reader may have noticed that the list reversal function above does not in fact reverse lists entirely in place. When the user calls **reverse**, the **Nil** argument in the call to **rev** will cause a new list cell to be allocated. Also, the **Nil** value at the end of the input list occupies a diamond, and this is simply discarded in the second line of the **rev** function (and will be subject to garbage collection if there are no other references to it). The overall effect is that we create a new diamond before calling the **rev** function and are left with an extra diamond after the call had completed. We could recover the extra diamond by making the **reverse** function return a pair consisting of the reversed list and the spare diamond, but this is rather clumsy and programs quickly become very complex when using this kind of technique. Camelot allows us to avoid this problem by writing

```
let rev l acc = match l with
    Nil@_ => acc
  | Cons (h,t)@d => rev t (Cons (h,acc)@d)

let reverse l = rev l Nil
```

The annotation **@_** causes the list cell to be placed on the freelist for later use.

The question now is how the user retrieves a diamond from the freelist. In fact our earlier description of construction invocation was incomplete. What actually happens is that if a program uses an undecorated constructor such as **Nil** or **Cons(4,Nil)** then the space required is only allocated on the heap if the freelist is empty; otherwise, a diamond is removed from the head of the freelist and is used to construct the value.

It may occasionally be useful to explicitly return a diamond to the freelist. This can be done by using the operator **free: ✧ -> unit**. For example, one might have a program fragment similar to the following:

```
... match l with
      Cons(h,t)@d =>
         let t' = f t
         in if h mod n = 0
            then t'
            else Cons(h,t')@d
```

The diamond **d** is not used in the left-hand branch of the **if** statement and is hence lost; we can make sure that the diamond is available for recycling by writing

```
         ...
         in if h mod n = 0
            then let _ = free d in t'
            else Cons(h,t')@d
```

There is one final notational refinement. The in-place list reversal function above is still not entirely satisfactory. If the freelist is empty when **reverse** is called then a new diamond will be allocated on the heap to contain the **Nil** value which is used for the initial accumulator (and the diamond occupied by the **Nil** at the end of the input list will be returned to the freelist at the end of the function call). Since diamond objects are represented by means of a JVM class the Java **null** value could be used to represent an empty list and this representation could then be used to simplify the reversal function. To achieve this, in each datatype declaration the programmer is allowed to prefix the name of a single zero-argument constructor with an exclamation mark. This directs the compiler to represent that constructor by **null** (and any attempt to re-use the space occupied by that constructor results in a compile-time error). The list-reversal example can now be rewritten as follows:

```
type intlist = !Nil | Cons of int * intlist

let rev l acc = match l with
    Nil => acc
  | Cons (h,t)@d => rev t (Cons (h,acc)@d)

let reverse l = rev l Nil
```

This now performs true in-place reversal. No heap space is consumed (or destroyed) when the **reverse** function is applied.

3.3. Object-oriented Camelot

* *This section should discuss the object-oriented features of Camelot, best by using the code for the PDA example program. First start with basics, by example, then the PDA program. Maybe a screenshot of that great application?*

Our main motivation for adding object-oriented features to Camelot is that we wish to create a system allowing inter-operation with Java, and that we wish to compile an object system to JVML. We therefore make the essential features of Java objects visible in Camelot in a simple form, without providing mechanisms of defining new classes on Camelot level.

3.3.1. Basic Features

We shall view objects as records of possibly mutable fields together with related methods, although Camelot has no existing record system. We define the usual operations on these objects, namely object creation, method invocation, field access and update, and casting and matching. As one might expect we choose a class-based system closely modelling the Java object system. Consequently we must acknowledge Java's uses of classes for encapsulation, and associate static methods and fields with classes also.

We now consider these features. The examples below illustrate the new classes of expressions we add to Camelot.

Static method calls:

> There is no conceptual difference between static methods and functions, ignoring the use of classes for encapsulation, so we can treat static method calls just like function calls: **java.lang.Math.max a b**

Static field access:

> Some libraries require the use of static fields. We should only need to provide access to constant static fields, so they correspond to simple values: **java.math.BigInteger.ONE**

Object creation:

> We clearly need a way to create objects, and there is no need to deviate from the \texttt{new} operator. By analogy with standard Camelot function application syntax (i.e. curried form) we have: **new java.math.BigInteger "101010" 2**

Instance field access:

> To retrieve the value of an instance variable, we write **object#field** whereas to update that value we use the syntax **object#field <- value** assuming that **field** is declared to be a *mutable* field.
>
> It could be argued that allowing unfettered external access to an object's variables is against the spirit of OO, and more to the point inappropriate for our small language extension, but we wish to allow easy interoperability with any external Java code.

Method invocation:

> Drawing inspiration from the O'Caml syntax, and again using a curried form, we have instance method invocation: **myMap#put key value**

Null values:

> In Java, any method with object return type may return the **null** object. For this reason we add a construct **isnull e** which tests if the expression **e** is a null value.

Casts and typecase:

> It may be occasionally be necessary to cast objects up to superclasses, for example to force the intended choice between overloaded methods. We will

also want to recover subclasses, such as when removing an object from a collection. Here we propose a simple notation for up-casting: **obj :> Class**

This notation is that of O'Caml, also borrowed by MLj (described in \cite{MLj}). To handle down-casting we shall extend patterns in the manner of \texttt{typecase} (again like MLj) as follows:

```
match obj with o :> C1 -> o.a()
             | o :> C2 -> o.b()
             | _ -> obj.c()
```

Here **o** is bound in the appropriate subexpressions to the object **obj** viewed as an object of type **C1** or **C2** respectively. As in datatype matches we require that every possible case is covered; here this means that the default case is mandatory. We also require that each class is a subclass of the type of **obj**, and suggest that a compiler warning should be given for any redundant matches.

Unlike MLj we choose not to allow downcasting outside of the new form of **match** statement, partly because at present Camelot has no exception support to handle invalid down-casts.

As usual, the arguments of a (static or instance) method invocation may be subclasses of the method's argument types, or classes implementing the specified interfaces.

The following example demonstrates some of the above features, and illustrates the ease of interoperability. We will discuss the need for type constraints as on the parameter **l** later.

```
let convert (l: string list) =
  match l with [] -> new java.util.LinkedList ()
             | h::t ->
               let ll = convert t
               in let _ = ll#addFirst h
               in ll
```

A more detailed example of a program using the object oriented features of Camelot can be found in Section 5.

3.4. Advanced Programming Techniques

* *These should be remarks on how to write safe programs when explicitly using at-notation.*

4. A more detailed example of Camelot in action

The following Camelot code takes a list xs of integers as input and replaces the first occurence of x with a y:

```
(* substitute the first occurence of y with x; IN-PLACE *)
let subst1 x y xs = match xs with [] -> []
                               | (z::zs)@d -> if (y=z)
                                              then (x::zs)@d
                                              else (z::(subst1 x y zs))@d
```

Note the @ annotation in the match branches of the code: on the left hand side, in a matching occurence, the annotation @d gives the name d to the construct being matched, in this case the current cons cell; on the right hand side, in a constructing occurence, this name is used to construct the result cell. Thus this code destructively updates the current cons cell, either replacing the head with x, if the value y has been found, or re-using the same head value, but replacing the tail with a reference to the list produced by a recursive call.

The following, purely functional, code creates a list of lists, by appending the first list to each of the elements of the second list.

```
(* map_cons xs = map (\ x -> (cons x xs)) *)
let map_cons xs ys = match ys with [] -> []
                                 | (h::t) -> (h::xs)::(map_cons xs t)
```

Note, that the above version will implicitly share the list ys in the tails of all list elements of the result list. If we want to generate a result list without such sharing we should use a version that implicitly clones ys.

```
let map_cons_clone xs ys = match ys with [] -> []
                                 | (h::t) -> (h::(clone_list xs))::(map_cons_clone xs t)
```

As long as the list elements are basic values we can use the following flat cloning function.

```
let clone_list xs = match xs with [] -> []
                                | (x'::xs') -> x'::(clone_list xs')
```

If the list elements are compound data types, such as lists, we have to do a deep cloning.

```
let deep_clone_list xs = match xs with [] -> []
                                     | (x'::xs') -> (clone_list x')::(deep_clone_list xs')
```

The following wrapper code demonstrates the behaviour of these destructive operations.

```
     let xs = (stringList_to_intList args)
  in let   _ = print_string "\n0. Map cons"
  in let l1  = xs
  in let l2  = (21::22::[])
  in let _   = print_string ("\nInput list:\n l1 = " ^ (show_list l1))
  in let _   = print_string ("\nInput list:\n l2 = " ^ (show_list l2))
  in let l3  = map_cons l2 l1
  in let _   = print_string ("\nmap_cons l2 l1:\n l3 = " ^ (show_listlist l3))
  in let l4  = clone_list l3
  in let _   = print_string ("\nclone of l3:\n l4 = " ^ (show_listlist l4))
  in let l9  = deep_clone_list l3
  in let _   = print_string ("\ndeep clone of l3:\n l9 = " ^ (show_listlist l9))
  in let l6  = map_cons_clone l2 l1
  in let _   = print_string ("\nmap_cons_clone l2 l1:\n l6 = " ^ (show_listlist l6))
  in let l5  = subst1 99 21 l2
  in let _   = print_string ("\nsubst1 99 21 l2 =  \n l5 = " ^ (show_list l5))
  in let _   = print_string ("\nl3 should be unchanged :\n l3 = " ^ (show_listlist l3))
  in let _   = print_string ("\nclone of l3: \n l4 = " ^ (show_listlist l4))
  in let _   = print_string ("\ndeep clone of l3:\n l9 = " ^ (show_listlist l9))
  in let _   = print_string ("\nl6 should be unchanged :\n l6 = " ^ (show_listlist l6))
```

Below is the result from running the program. First, we generate several lists to work with.

```
--(romulus[140](2.05b))-- java testList3 3 4 5

0. Map cons
Input list:
 l1 = [3, 4, 5]
Input list:
 l2 = [21, 22]
map_cons l2 l1:
 l3 = [[3, 21, 22], [4, 21, 22], [5, 21, 22]]
clone of l3:
 l4 = [[3, 21, 22], [4, 21, 22], [5, 21, 22]]
deep clone of l3:
 l9 = [[3, 21, 22], [4, 21, 22], [5, 21, 22]]
map_cons_clone l2 l1:
 l6 = [[3, 21, 22], [4, 21, 22], [5, 21, 22]]
```

Note that l3, l4, l6 and l9 are constructed to denote the same lists, but internally they have different representations. This shows up when we substitute 99 for the first occurence of 21 in l2.

```
subst1 99 21 l2 =
```

```
 l5 = [99, 22]
```

The value of the result list is as expected. But remember that we used l2 in constructing l3, doing a map_cons. Therefore, the contents of l3 has changed, too.

```
l3 should be unchanged :
 l3 = [[3, 99, 22], [4, 99, 22], [5, 99, 22]]
```

Note that all functions involved in generating l3 have been purely functional. The only impure functions, subst1, was just applied to l2, and caused a change in l3. The goal in Camelot, is to use high-level type systems, which prohibit such code (e.g. linear types). Without such security on the type-system level, however, global changes like this are possible.

You might think that l4 remained unchanged, since it is a clone of the original l3. However, the clone function only clones the spine of the list, but it still shares the list elements. Therefore, l4 has changed too.

```
clone of l3:
 l4 = [[3, 99, 22], [4, 99, 22], [5, 99, 22]]
```

A deep clone of the original l3, which also clones the list elements, remains unchanged. The same holds for l6, which was generated by a map_cons function that implicitly does cloning.

```
deep clone of l3:
 l9 = [[3, 21, 22], [4, 21, 22], [5, 21, 22]]
l6 should be unchanged :
 l6 = [[3, 21, 22], [4, 21, 22], [5, 21, 22]]
```

Obviously, using unrestricted forms of in-place operations is dangerous and can lead to surprising side-effects. However, the type system can ensure that no such dangerous operations are used. The simplest form is a purely linear type system that stipulates that each variable is used just once. Using linear type checking therefore rejects the program we have written.

* *Add results from running linear type checking. Add a subsection on more refined type systems and work in progress.*

5. An Example of Object-Oriented Features

Here we give an example of using the object-oriented features in Camelot. The code below, together with the two standard utility functions **rev** and **len** for list reversal and length, defines a program for Sun's MIDP platform (as described in midp), which runs on devices such as PalmOS PDAs. The program displays the list of primes in an interval. Two numbers are entered into the first page of the GUI, and when a button is pressed a second screen appears with the list of primes, calculated using the sieve of Eratosthenes, along with a button leading back to the initial display.

This example has been compiled with our current compiler implementation, and executed on a PalmOS device.

```
class primes = javax.microedition.midlet.MIDlet with
  implement javax.microedition.lcdui.CommandListener

  field exitCommand: javax.microedition.lcdui.Command
  field goCommand: javax.microedition.lcdui.Command
  field doneCommand: javax.microedition.lcdui.Command
  field mainForm: javax.microedition.lcdui.Form
  (* lower and upper limits: *)
  field lltf: javax.microedition.lcdui.TextField
  field ultf: javax.microedition.lcdui.TextField
  field display: javax.microedition.lcdui.Display

  maker () =
    let _ = display <-
      (javax.microedition.lcdui.Display.getDisplay
        (this:> javax.microedition.midlet.MIDlet))
    in let _ = goCommand <-
      (new javax.microedition.lcdui.Command
      "Go" javax.microedition.lcdui.Command.SCREEN 1)
    in let _ = exitCommand <-
      (new javax.microedition.lcdui.Command
      "Exit" javax.microedition.lcdui.Command.SCREEN  2)
    in let t = new javax.microedition.lcdui.Form "Primes"
    in let ll = new javax.microedition.lcdui.TextField
        "Lower limit:" "" 10
        javax.microedition.lcdui.TextField.NUMERIC
    in let _ = lltf <- ll
    in let _ = t#append ll
    in let ul = new javax.microedition.lcdui.TextField
        "Upper limit:" "" 10
        javax.microedition.lcdui.TextField.NUMERIC
    in let _ = ultf <- ul
    in let _ = t#append ul
    in let _ = t#addCommand (this#goCommand)
    in let _ = t#addCommand (this#exitCommand)
    in let _ = mainForm <- t
    in t#setCommandListener this

  method startApp (): unit =
    this#display#setCurrent (this#mainForm)
```

```
  method pauseApp (): unit = ()
  method destroyApp (b:bool): unit = ()
  method commandAction
           (cmd: javax.microedition.lcdui.Command)
           (s: javax.microedition.lcdui.Displayable)
           : unit =
    if cmd#equals (this#exitCommand)
    then let _ = this#destroyApp false
         in this#notifyDestroyed ()
    (* create & display list of primes *)
    else if cmd#equals (this#goCommand)
    then
      let    lower_limit = int_of_string
                            (this#lltf#getString())
      in let upper_limit = int_of_string
                            (this#ultf#getString())
      in let primes =
               new javax.microedition.lcdui.Form "Primes"
      in let _ = appendPrimes lower_limit upper_limit primes
      in let done = new javax.microedition.lcdui.Command
                 "Done"
                 javax.microedition.lcdui.Command.SCREEN 1
      in let _ = doneCommand <- done
      in let _ = primes#addCommand done
      in let _ = primes#setCommandListener this
      in let _ =
        javax.microedition.lcdui.AlertType.INFO#playSound
          (this#display)
      in this#display#setCurrent primes
   (* back to main form *)
    else if cmd#equals (this#doneCommand) then
      this#display#setCurrent (this#mainForm)
    else ()
end
(* Generate a list of prime numbers in an interval [a..b] *)
(* Integer square roots *)
let increase k n = if (k+1)*(k+1) > n then k else k+1
let rec intsqrt n = if n = 0 then 0
                    else increase (2*(intsqrt (n/4))) n

(* n is divisible by no member of l which is <= sqrt n *)
let isPrime n l lim =
   match l with
     [] -> true
   | h::t -> h <= lim && n mod h <> 0 && isPrime n t lim

(* generate list of primes between n and top *)
let make1 n top acc =
```

```
  if n > top then rev acc []
  else if isPrime n acc n then make1 (n+2) top (n::acc)
  else make1 (n+2) top acc

let makeSmallPrimes top = make1 3 top [2]
let makePrimes n top smallPrimes =
  if n > top then []
  else if isPrime n smallPrimes n then
    n::(makePrimes (n+2) top smallPrimes)
  else makePrimes (n+2) top smallPrimes

let appList l (f: javax.microedition.lcdui.Form) =
  match l with [] -> ()
  | (h::t)@_ -> let _ = f#append ( (string_of_int h)^"\n")
                in appList t f

let appendPrimes bot top
      (f: javax.microedition.lcdui.Form) =
  let smallPrimes = makeSmallPrimes (intsqrt top)
  in let primes = makePrimes (bot + 1 - bot mod 2)
                    top smallPrimes
  in let s = (string_of_int (len primes)) ^ " primes\n"
  in let _ = f#append s
  in appList primes f
```

6. Camelot the System

6.1. Compiler options

As of version 3.1 of the Camelot compiler the following compile time options are available.

-h, -help

Output this help message.

-a1, -absyn1

Print normalised abstract syntax.

-a2, -absyn2

Print monomorphised normalised abstract syntax.

-ab, -asdl-bin

Like -a2 but write the syntax to an ASDL binary pickle file (.asdl.bin).

-as, -asdl-sexp

Like -ab but write an ASDL S-expression pickle file (.asdl.sexp).

-b, -debug

Debug mode. Implies verbose. Gives some probably useless information. Pages of output per function, mainly type inference and monomorphisation related.

-c, -classnames

Rather than saying \"compiled ...\", just print out the names of the classes which have been generated (this was for use in the online demo)

-cp `dirlist`, -classpath `dirlist`

This option is for use with object-oriented Camelot extensions. <dirlist> consists of a colon-separated list of directory names in which the compiler will look for JVM classes (including the standard Java libraries). If the compiler needs to look for classfiles and no -cp option is given then directories listed in the environment variable CAMELOT_CLASSPATH will be used; if this variable is not set then the compiler will use ${HOME}/classes:. . If you're using the shell variable, don't put a ~ in any of the paths: use $HOME instead. See also the -j and -m options.

-d `dir`

Write the output classes into directory `dir`.

-D, -diamond-layout

Print information about structure of diamond class

-f, -freelist

Print debugging information about behaviour of freelist allocation during execution.

-g, -grail

Output Grail to .gr file (no class file)

-grail-cases

Compile Camelot matches using the experimental Grail case construct

-hints, -proof-hints

Output a file ...-HINTS.asdl.bin containing proof hints

-j, -java-rt

Same as -cp /group/project/mrg/java_rt_sun-1.4.1:. (For compiling with the standard Java class hierarchy on DICE machines in Edinburgh)

-l, -linear

Turn on linearity checking

-lfd

Create .lfd file and call lfd_infer

-ll, -layered

Turn on layered sharing checking and inference (needs deel_uapl_prg)

-lu, -usage-aspects, -readonly

Turn on usage aspects checking and inference (needs deel_uapl_prg) (NOT YET),

-lb, -debug-usage

Debug the layered sharing and usage aspect inferences

-m, -midp

Same as -cp /group/project/mrg/midp-1.0.3_classes:. (Useful for compiling with the MIDP 1.0 API on DICE machines in Edinburgh)

-N, -nodiamonds

Use the original datatype scheme: one Java class for each datatype (may not work any more).

-n, -no-tail-call-elimination

Turn off tail-call elimination.

-no-consolidation

Turn off Grail local variable consolidation.

-od, -optimise-destruction, -auto-destruction

Turn on destruction inference (works only with -ld or -lu) (NOT YET).

-r, -record-freelist, --record-freelist

Record information about the freelist. Programs can recover this information by calling diamond_info: unit -> string

-readable

Omit argument lists from Grail output: the program probably won't execute, but it's easier to read.

-s, -silent

Silent mode. Suppress all textual output when compiling correct programs.

-show-all

Show adpator functions when printing abstract syntax.

-t, -time

Print the current date and time

-thy,

Emit Isabelle theory files

-v, -verbose

Verbose mode. Does nothing.

-x, -exit

Exit from compiler immediately

* *Add stuff about how to get performance results from the program. Use off-the-shelf JVM performance monitoring tools.*

6.2. Hints on Performance Tuning

* *General hints on what to do and not to do in Camelot programs.*

6.3. Internals of compilation and execution

* *Some important internals relevant for the user (uniform diamond representation, mapping of data types to classes etc.)*

7. Bibliography

Bibliography entries.

A. Appendix

A.1. Camelot Syntax

* *This is currently taken out of* `Parser.grm`*. Do we have an up-to-date, vanilla version of the Camelot syntax around?*

```
exp:
  IF exp THEN exp ELSE exp
| MATCH exp WITH barOpt match
| infixExp
| letExp
| infixExp ATSIGN identl
;


infixExp:
  atomExp
| ident atomExpRow
| ExtId atomExpRow
| infixExp HASH ident atomExpRow
| infixExp HASH ident
| NEW ExtId atomExpRow
| NEW ident atomExpRow
| ISNULL atomExp
| SUPER atomExpRow
| ConId expSeq
| infixExp CONS infixExp
| infixExp PLUS infixExp
| infixExp MINUS infixExp
| infixExp STAR infixExp
| infixExp SLASH infixExp
| infixExp MOD infixExp
| infixExp PLUSDOT infixExp
| infixExp MINUSDOT infixExp
| infixExp STARDOT infixExp
| infixExp SLASHDOT infixExp
| MINUS atomExp %prec UMINUS
| MINUSDOT atomExp %prec UMINUS
```

```
    | infixExp EQUALS infixExp
    | infixExp DIAMOND infixExp
    | infixExp LESS infixExp
    | infixExp GREATER infixExp
    | infixExp LTEQ infixExp
    | infixExp GTEQ infixExp
    | infixExp EQUALSDOT infixExp
    | infixExp DIAMONDDOT infixExp
    | infixExp LESSDOT infixExp
    | infixExp GREATERDOT infixExp
    | infixExp LTEQDOT infixExp
    | infixExp GTEQDOT infixExp
    | infixExp AMP     infixExp
    | infixExp AMPAMP  infixExp
    | infixExp OR      infixExp
    | infixExp BARBAR  infixExp
    | infixExp CARET infixExp
    | NOT atomExp
    | infixExp COLON type
    | infixExp COERCE type
    | ident UPDATE infixExp
    ;

    atomExp:
      intval
    | floatval
    | ConId
    | LSQ RSQ
    | LSQ listExpSeq RSQ
    | ident
    | ExtId
    | quoteval
    | charval
    | TRUE
    | FALSE
    | BEGIN  exp END
    | LPAREN exp RPAREN
    | UNIT
    | NULL COERCE ExtId
    ;

    listExpSeq:
      exp
    | exp SEMI listExpSeq
    ;

    letExp:
      LET ident EQUALS exp IN exp
```

```
| LET USCORE EQUALS exp IN exp
;

expSeq:
  LPAREN expSeq1 RPAREN
| UNIT
;

expSeq1:
  exp
| exp COMMA expSeq1
;

atomExpRow:
  atomExp
| atomExp atomExpRow
;

match:
  matchcase BAR match
| matchcase   %prec ARROW
;

matchcase:
  ConId patSeq ARROW exp
| ConId patSeq ATSIGN USCORE ARROW exp
| ConId patSeq ATSIGN identl ARROW exp
| LSQ RSQ ARROW exp
| LSQ RSQ ATSIGN USCORE ARROW exp
| pat CONS pat ARROW exp
| LPAREN pat CONS pat RPAREN ARROW exp
| LPAREN pat CONS pat RPAREN ATSIGN USCORE ARROW exp
| LPAREN pat CONS pat RPAREN ATSIGN identl ARROW exp
| pat COERCE anyId ARROW exp
| USCORE ARROW exp
;

patSeq:
  LPAREN patSeq1 RPAREN
| /* empty */
;

patSeq1:
  pat
| pat COMMA patSeq1


pat:
```

```
  USCORE
| ident
;

typeDecSeq:
  typeDec typeDecSeq
| /* empty */
;

typeDec:
  TYPE tVarSeq typeName EQUALS constrList
;

tVarSeq:
  TYVAR
| LPAREN tVarSeq1 RPAREN
| /* empty */
;

tVarSeq1:
  TYVAR
| TYVAR COMMA tVarSeq1
;

constrList:
  constr constrListOpt
| /* empty */
;

constrListOpt:
  BAR constr constrListOpt
| /* empty */
;

constr:
  ConId
| BANG ConId
| ConId OF prodtype
;

prodtype:
  type
| type STAR prodtype
;

identl:
   ident
;
```

```
anyId:
   ident
|  ExtId
;

identSeq:
   /* empty */
| ident identSeq
;

valDec:
  VAL anyId COLON type
| CLASSTYPE anyId superOpt1 COLON implementSeq classValDecSeq END
;

classValDec:
  valDec
| instanceValDec
;

valDecSeq:
  valDec valDecSeq
| /* empty */
;
classValDecSeq:
  classValDec classValDecSeq
| /* empty */
;

funBlockSeq:
  funBlock funBlockSeq
| /* empty */
;

funBlock:
  LET recOpt funDef funBlockOpt
;

funBlockOpt:
  AND funDef funBlockOpt
| /* empty */
;

funDef:
  ident farglist EQUALS exp
;
```

```
recOpt:
  REC
| /* empty */
;

barOpt:
  BAR
| /* empty */
;

farglist:
  farglist1
;

farglist1:
  fargitem
| fargitem farglist1
;

fargitem:
  ident
| LPAREN RPAREN
| UNIT
| LPAREN fargitem RPAREN
| LPAREN ident COLON type RPAREN
;

classImp:
  CLASS classId identSeq EQUALS superArgsOpt implementSeq ciValDecSeq methodImpSeq END
;
classId:
   ident
|  ConId
|  ExtId
;

classImpSeq:
  /* empty */
| classImp classImpSeq
;

ciValDec:
  valDec
| fieldValDec
;
ciValDecSeq:
  ciValDec ciValDecSeq
| /* empty */
```

```
;

superArgsOpt:
  /* empty */
| anyId identSeq WITH
;

superOpt1:
  /* empty */
| IS anyId
;
implementSeq:
  /* empty */
| implement implementSeq
;
implement:
  IMPLEMENT anyId
;
methodImpSeq:
  /* empty */
| methodImp methodImpSeq
;

methodImp:
  METHOD ident farglist tyOpt EQUALS exp
| MAKER farglist superMakerOpt EQUALS exp
| LET recOpt ident farglist tyOpt EQUALS exp /* funImpOpt */
;

superMakerOpt:
  /* empty */
| COLON SUPER atomExpRow
;

tyOpt:
  /* empty */
| COLON type
;

type:
  type1 ARROW type
| type1
;

type1:
  INT
| CHAR
| BOOL
```

```
  | FLOAT
  | STRING
  | UNITT
  | TYVAR
  | DIAMOND
  | ExtId
  | type1 ARRAY
  | fancyTypeName
  | LPAREN type RPAREN
  ;

fancyTypeName:
    typeName
  | type1 typeName
  | LPAREN typeSeq RPAREN typeName
  ;

instanceValDec:
    methodValDec
  | fieldValDec
  ;

methodValDec:
    METHOD ident COLON type
  ;

fieldValDec:
  FIELD ident COLON type
  ;

typeSeq :
    type1 COMMA type1
  | type1 COMMA typeSeq
  ;

typeName:
    ident
  ;
```

A.2. Built-in types, operators and functions

* *This appendix should containt the Camelot Prelude, i.e. the types and specifications of the functions contained in* `Camelotlib.java`*. Additionally, we should identify important functions we want to have around all the time and put this in such a standard library, too (List; take,drop,...)*

Camelot has a built-in polymorphic list type α **list** . The empty list (which is represented by the **null** value) is denoted by **[]** and the cons operation is denoted by **::**. For example, to define the list length function (which is not presently built in) one can write

```
let length l =
    match l with
          [] => 0
        | h::t => 1+length t
```

Literal lists can also be represented by the syntax used in SML. For example

```
let n = let l = [1,5,7,9,33] in length l
```

This syntax is not allowed in match statements.

A.2.1. Operators

Camelot has various built-in infix operators.

+, -, *, /, mod

arithmetic operations on integers

+., -., *., /.

arithmetic operations on floating-point values

=, <, <=, =>, >

numeric comparisons.

Note that the equality operator = can also be applied to strings and other objects. However, it is interpreted as equality of references and hence will usually fail to give the correct result; for strings, one should use the function **same_string**.

not, andalso, orelse

boolean operators.

The last two of these only evaluate as many expressions as are required to decide the result; for example, if we attempt to evaluate the expression **e1 andalso e2** then e2 will not be evaluated if e1 evaluates to true.

^

string concatenation.

A.2.2. Functions

There are also several built-in functions. The names of most of these should be self-explanatory.

```
int_of_float: float -> int
float_of_int: int -> float
int_of_string: string -> int
string_of_int: int -> string
float_of_string: string -> float
string_of_float: float -> string
print_int: int -> unit
print_int_newline: int -> unit
print_float: float -> unit
print_float_newline: float -> unit
print_string: string -> unit
print_string_newline: string -> unit
print_newline: unit -> unit
same_string: string -> string -> bool
(* Array operations *)
empty: int -> α -> α array
get:   α array -> int -> α
set:   α array -> int -> unit
arraylength: α array -> int
```

A.3. Camelot Example Programs

A.3.1. List Examples

* *Taken from* `List.cmlt`.

```
let gen a b = if a = b+1 then []
              else a::(gen (a+1) b)
(* Generate list [a..b] *)

let len l = match l with [] -> 0 | h::t -> 1 + len t
let hd l = match l with [] -> 0  | h::_ -> h
let tl l = match l with [] -> [] | _::t -> t
```

```
let append l m =
  match l with [] -> m
             | h::t -> h::(append t m)

let rev' l acc =
  match l with [] -> acc
             | h::t -> rev' t (h::acc)

let rev l =
  match l with [] -> []
             | h::t -> append (rev t) [h]

let take l n = match l with  [] -> []
                         | h::t -> if n > 0
                                   then h::(take t (n-1))
                                   else []

let drop l n = match l with [] -> []
                          | h::t -> if n > 0 then drop t (n-1)
               else l

(* Insertion sort *)
let insert n l =
  match l with [] -> [n]
             | h::t -> if n <= h then n::l
                       else h::(insert n t)
let sort l =
  match l with [] -> []
             | h::t -> insert h (sort t)

(* Quick sort (from Paulson) *)
let qsort l =
  match l with
  | [] -> []
  | h::t ->
      begin
        match t with
 | [] -> l
       | x::y -> partition h [] [] t
      end

and partition a left right l =
  match l with [] -> append (qsort left) (a::(qsort right))
             | x::xs ->
               if x <= a then partition a (x::left) right xs
               else partition a left (x::right) xs
```

```
(* Merge sort. Again from Paulson *)
let merge lx ly =
  match lx with [] -> ly
              | x::xs ->
    begin
                    match ly with [] -> lx
                    | y::ys ->
                        if x <= y
                        then x::(merge xs (y::ys))
                        else y::(merge (x::xs) ys)
    end

let mergesort l =
  match l with [] -> []
             | h::t ->
   begin
     match t with
       [] -> l
                   | _::_ ->
         let k = (len l)/2
         in
         merge (mergesort (take l k)) (mergesort (drop l k))
   end

let member n l =
  match l with [] -> false
             | h::t -> if n = h then true
                       else member n t


let listToString' l =
        match l with
          [] -> ""
        | h::t ->
     begin
       match t with [] -> string_of_int h
              | h'::t' -> (string_of_int h)
              ^ ", "
              ^ (listToString' t)
     end

let listToString l =
           "[" ^ (listToString' l) ^ "]"


let stringListToIntList l = match l with
    [] -> []
  | h::t -> (int_of_string h) :: (stringListToIntList t)
```

```
let sq x = x*x
let treble x = x+x+x

let start args =
 let l = stringListToIntList args
     in let _ = print_string ("List has length "
                             ^ string_of_int(len l)
                             ^"\n")
     in let _ = print_string ("Original list: "
                             ^ (listToString l)
                             ^ "\n")
     in let _ = print_string ("Reversed list: "
                             ^ (listToString (rev' l []))
                             ^ "\n")
     in let _ = print_string ("First 4 elements of list: "
                             ^ (listToString (take l 4))
                             ^ "\n")
     in let _ = print_string ("All but first 4 elements of list: "
                             ^ (listToString (drop l 4))
                             ^ "\n")
     in let _ = print_string ("Insertion sorted list:   "
                              ^ (listToString (sort l))
                              ^ "\n")
     in let _ = print_string ("Quick sorted list:   "
                              ^ (listToString (qsort l))
                              ^ "\n")
     in let _ = print_string ("Merge sorted list:   "
                              ^ (listToString (mergesort l))
                              ^ "\n")

     in ()
```

A.3.2. Sorting Algorithms

A.3.2.1. Insertion Sort

```
type iList = !Nil | Cons of int * iList

let ins a l =
 match l with
   Nil -> Cons(a,Nil)
 | Cons(x,t)@_ ->
     if a < x then Cons(a,Cons(x,t))
  else Cons(x, ins a t)
```

```
let sort l =
  match l with
    Nil -> Nil
  | Cons(a,t)@_ -> ins a (sort t)
```

A.3.2.2. Heap Sort

```
(* Linear in-place heapsort code for database program *)

type iList = !Nil | Cons of int * iList
type iresult = !None | Some of int * iTree
type iTree = !Leaf | Node of int * iTree * iTree

(* Message after copy_strings *)
(* Odd error if you use -n option.  item seems to be assigned two
   types:  Object and Item.  This appears to be reolved in DataFlow.sml,
   but if we turn this off then we get a Grail error. *)

let string_less v w = string_compare v w <= 0

(* all trees are supposed to be heaps. Leaf is a heap and
t=Node(w,t1,t2) is a heap if w is the largest element of t and
moreover t1, t2 are heaps and 0<=|t1|-|t2|<=1 *)

(* insert a new element into a heap *)

let insert x t =
 match t with
        Leaf ->  Node(x,Leaf,Leaf)
      | Node(z,left,right)@_ ->
             if x < z
                then Node(x, insert z right, left)
                else Node(z, insert x right, left)

(* siftdown w t1 t2 assumes that t1, t2 are heaps and that 0 <= |t1|-|t2| <= 1.
   It returns a heap consisting of the elements of Node(w,t1,t2) *)

let siftdown w t1 t2 =
 match t1 with Leaf -> Node(w,Leaf,Leaf)
      | Node(v,t11,t12)@_ ->
 begin
    match t2 with
           Leaf ->
                if w < v then
                     Node(w, Node(v,Leaf,Leaf), Leaf) else
```

```
                  Node(v, Node(w,Leaf,Leaf), Leaf)
         | Node(u, t21,t22)@_ ->
              if w < u & w < v then
                    Node(w, Node(v,t11,t12), Node(u,t21,t22))
              else if u < w & u < v then
                    Node(u, Node(v,t11,t12), siftdown w t21 t22)
              else  Node(v, siftdown w t11 t12, Node(u,t21,t22))
        end

(* removes an arbitrary element from t and returns it as well as the
   resulting heap *)

let removesome t =
    match t with Leaf -> None
  | Node(x, left, right)@_ ->
      begin
        match removesome left with
          None -> Some(x, right) (* actually, right=Leaf, here *)
        | Some(z, left')@_ -> Some(z, Node(x, right, left'))
      end


(* removes the largest element from a heap *)

let removetop t =
    match t with
    Leaf -> None
 |  Node(x,left,right)@_ ->
     begin
       match removesome left with
  None -> Some(x, right) (* actually, right=Leaf here *)
       | Some(z, left')@_ -> Some(x, siftdown z right left')
     end


let make_heap l = match l with Nil -> Leaf
|  Cons(h, t)@_ ->
    let hp = make_heap t
    in insert h hp

let extract h = match removetop h with
                 None -> Nil
               | Some(h,t)@_ -> Cons(h, extract t)


let sort l = extract (make_heap l)
```