Final Honour School of Computer Science
Final Honour School of Mathematics and Computer Science
Degree of Master of Science in Computer Science

Priniciples of Programming Languages

Laboratory Manual

Mike Spivey

Oxford University Department of Computer Science
Draft of 12.x.2013

Contents

Introduction

This manual describes the practical component of the third year course 'Principles of Programming Languages'. The course includes three laboratory exercises, all based on the language *Fun*:

- In the first exercise, you will write some programs in a dialect of *Fun* that has assignable variables, comparing functional and imperative styles of programming.
- In the second exercise, you will implement a variant of *Fun* that has assignable variables and parameters passed by name, investigating the interaction of these two features.
- In the third exercise, you will investigate the semantics of a loop-with-exit construct that generalizes one found in Oberon.

I suggest you attempt all three exercises if you have time, or treat the first one as optional if you do not find it helpful.

The lab materials mentioned in this manual are delivered using the version control system Subversion, so that any fixes to the lab code or examples that are made during term can be merged with your work easily and automatically. The next section explains how to set up the materials for the labs under your account on the lab machines. The materials can also be found on the course wiki at

    http://spivey.oriel.ox.ac.uk/corner/Programming_Languages

In these labs, we will work with interpreters written in Haskell, but in place of the *hugs* interpreter that you have probably used in the past, we shall be using the Glasgow Haskell Compiler *ghc*. Besides making our interpreters run faster by compiling them into machine code, *ghc* makes it easier to build a friendly top-level dialogue, including a history of previous commands. Using *ghc* means that the process of running a Haskell program is a little different: instead of starting *hugs* and loading our program into it, we first compile the Haskell program by using *ghc* in a UNIX command, and we obtain an executable program that can be run like any other compiled program. The details of how to do this are given in the lab instructions.

1 Setting up the labs

Materials for the labs are held in a Subversion repository on Mike Spivey's college machine, with read-only anonymous access. In order to check out a copy of the lab materials, you should give the command,

```
$ svn co svn://spivey.oriel.ox.ac.uk/labs/proglan
```

inter This will create a directory called proglan and populate it with the lab materials, with separate subdirectories proglan/lab[123] for each lab, and an additional subdirectory proglan/examples that contains code for all the interpreters that will be demonstrated in lectures.

Corrections may be made and further modules may be added during the course of the term. You can bring your copy of the lab materials up to date at any time by changing to the proglan directory and using the command

```
$ svn up
```

You can also use

```
$ svn diff
```

to see the changes you have made. Because the repository is read-only, there's no provision for you to check in your changes. Sorry!

2 Using your own machine

Chapter 1

Lab one: Functional vs Imperative

In this lab, you will write some functional programs in *Fun*, then translate one of them into an imperative program that uses a loop instead of recursion. This will allow you to become familiar with working in the syntax of *Fun*. The exercise also has an ulterior motive, because later (if there is time in the course) we shall use again the idea of transformating from functional to imperative form that is illustrated here. We will use this idea to show how our meta-circular interpreters for *Fun* (written in Haskell) can be transformed into a simpler language that has iteration instead of recursion.

1.1 Getting started

For this lab, I have provided an interpreter *FunLab1* that adds assignable variables and output to the basic *Fun* language. Despite these additions, you can still use *FunLab1* to run purely functional *Fun* programs, so you will be able to use the same interpreter throughout the lab.[1] The interpeter and some sample files of *Fun* code are kept in the lab directory `lab1`. You will find the following files:

`Parsing.hs`	Parsing library
`Environment.hs`	Implementation of environments
`Memory.hs`	Implementation of memories
`FunSyntax.hs`	Abstract syntax of *Fun*
`FunParser.hs`	Parser for *Fun*
`FunLab1.hs`	Interpreter
`examples.fun`	Example definitions
`facimp.fun`	Factorial implemented imperatively

In the lab, you will write a file `script.fun` containing a *Fun* program of your own.

The very first step in the lab is to build the *FunLab1* interpreter; since we will just be writing *Fun* programs in this lab, and not modifying the implementation of *Fun*, we need carry out this process only once. Having obtained

[1] The *FunLab1* interpreter is built using the technique of *monads* that we will not study until a little later in the course. You don't need to be concerned with this to use it, though.

a copy of the source files as above, you need only type one command for the whole process to happen:[2]

```
% ghc --make FunLab1.hs -main-is FunLab1 -o fun
Chasing modules from: FunLab1.hs
Compiling Parsing        ( Parsing.hs, ./Parsing.o )
Compiling FunSyntax      ( FunSyntax.hs, ./FunSyntax.o )
Compiling FunParser      ( FunParser.hs, ./FunParser.o )
Compiling Memory         ( Memory.hs, ./Memory.o )
Compiling Environment    ( Environment.hs, ./Environment.o )
Compiling Main           ( FunLab1.hs, ./FunLab1.o )
Linking ...
```

The Haskell compiler compiles each of the modules in the interpreter, then links them all together into an executable called fun.

Let's use this freshly-built interpreter to run some of the examples in the file examples.fun, listed in Figure 1.1. For comparison, similar functions are defined in Haskell in Figure 1.2.

```
% ./fun examples.fun
--- interval = <function>
--- map = <function>
--- sum = <function>
--- sqsum = <function>
>>>
```

Here we have started the program *fun*, and given as an argument the file examples.fun. The interpreter has loaded the file and printed the names of the four functions defined in it.

The first definition shows a recursive function being defined with the keyword rec:

```
rec interval(a, b) =    -- [a, a+1, ..., b]
  if a > b then nil else a : interval(a+1, b);;
```

This definition also contains the keywords if, then and else, and a comment begining with '--'. The result list is constructed using the primitive operator : and the constant nil. Each top-level phrase (in a file or at the interactive prompt) ends in ';;'.

The second definition illustrates how *Fun* allows higher-order functions:

```
rec map(f, xs) =
  if xs = nil then nil else f(head(xs)) : map(f, tail(xs));;
```

Here, a function f is passed as a parameter to map and called from its body. This function also shows the use of head and tail to take apart the argument list.

At the interactive prompt, we can type expressions to be evaluated:

```
>>> interval(1, 10);;
--> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> let val f(x) = x * x in map(f, interval(1, 10));;
--> [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

[2] In this and subsequent scripts of interaction with computer, the text that you type is shown in *italics*.

```
-- examples.fun

rec interval(a, b) =    -- [a, a+1, ..., b]
  if a > b then nil else a : interval(a+1, b);;

rec map(f, xs) =
  if xs = nil then nil else f(head(xs)) : map(f, tail(xs));;

val sum(xs) =
  let rec loop(ys, s) =
    if ys = nil then s else loop(tail(ys), s + head(ys)) in
  loop(xs, 0);;
```

Figure 1.1: *Contents of file* examples.fun

$interval\ a\ b = \textbf{if}\ a > b\ \textbf{then}\ [\,]\ \textbf{else}\ a : interval\,(a+1)\ b$

$map\,f\ [\,] = [\,]$
$map\,f\ (x : xs) = f\ x : map\,f\ xs$

$sum\ xs = loop\ xs\ 0$
  where
    $loop\ [\,]\ s = s$
    $loop\ (y : ys)\ s = loop\ ys\ (s + y)$

Figure 1.2: *Example definitions rewritten in Haskell*

```
>>> map(lambda (x) x * x, interval(1, 10));;
--> [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

The last of these three expressions shows the use of lambda to write an anonymous function: as you can see, it is equivalent to the second expression.

We can also add new definitions to the environment and use them in subsequent expressions, giving yet another way to compute a table of squares:

```
>>> val sqr(x) = x * x;;
--- sqr = <function>
>>> map(sqr, interval(1, 10));;
--> [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

In contrast, an expression beginning with let introduces a constant that is local to the expression, and does not become part of the top-level environment:

```
>>> let val n = 10 in sum(interval(1, n));;
--> 55
>>> n;;
Error: n is not declared
```

Finally, there is a primitive function list that is useful for testing other functions; list takes any number of arguments and returns a list (built with : and nil) that contains those arguments.

```
>>> list(1, 2, 3, 4);;
```

```
--> [1, 2, 3, 4]
>>> sum(list(3, 1, 4, 1, 5));;
--> 14
```

The same list could be written 1:2:3:4:nil, but the list function makes things a bit clearer, especially when nested lists become involved.

When you have finished interacting with the *Fun* interpreter, you can return to the UNIX prompt by typing Ctrl–D, i.e., holding down the Ctrl key while pressing D.

```
>>> Ctrl-D
Bye

%
```

It is necessary to exit like this and start the *Fun* interpreter again in order to make it notice any changes you have made to the files of definitions it initially loads.

The *FunLab1* interpreter implements a dialect of *Fun* that includes assignable variables:

```
% ./fun
>>> val a = new();;
--- a = <address 0>
>>> !a;;
Error: uninitialized location 0
>>> a := 3;;
--> 3
>>> !a;;
--> 3
>>> Ctrl-D
```

Another supplied file, facimp.fun, contains an imperative implementation of the factorial function:

```
-- Factorial implemented imperatively
val fac(n) =
  let val k = new() in
  let val r = new() in
  k := n; r := 1;
  while !k > 0 do
    (r := !r * !k; k := !k - 1);
  !r;;
```

Though it uses assignable variables, this function behaves just like the functional program for factorial:

```
% ./fun facimp.fun
--- fac = <function>
>>> fac(6);;
--> 720
```

1.2 Flattening nested lists

In this section, you will program some basic list-processing functions in *Fun*. You should put your definitions in a file with a name like script.fun, so that you can load them into the interpeter for testing.

Haskell has an operator ++ for appending lists. An equivalent function *append* might be defined as follows.

$$\begin{aligned}&append :: [a] \to [a] \to [a]\\ &append\ [\,]\ ys = ys\\ &append\ (x : xs)\ ys = x : append\ xs\ ys\end{aligned}$$

Define a similar function append in *Fun* and test it:

```
>>> append(list(1, 2, 3), list(4, 5, 6));;
--> [1, 2, 3, 4, 5, 6]
```

Haskell also has a function *concat* for concatenating a list of lists:

$$\begin{aligned}&concat :: [[a]] \to [a]\\ &concat\ [\,] = [\,]\\ &concat\ (xs : xss) = append\ xs\ (concat\ xss)\end{aligned}$$

Define concat in *Fun* and test it:

```
>>> val xxx = list(list(1, 2), nil, list(3, 4));;
--- xxx = [[1, 2], [], [3, 4]]
>>> concat(xxx);;
--> [1, 2, 3, 4]
```

Because Haskell is statically typed, lists of integers are different from lists of lists of integers, and it is not possible to mix the two. In *Fun*, however, there is no static typing, so it is possible to make a list in which some elements are integers and others are lists, like this:

```
>>> val yyy = list(1, list(2, 3), list(4, list(5), 6));;
--- yyy = [1, [2, 3], [4, [5], 6]]
```

(It will be convenient if you add this definition of yyy to your file of code, so that you can use it for testing various functions later.) If we want to reduce such a nested list to a simple list of integers, the simple function concat won't work:

```
>>> concat(yyy);;
Error: bad arguments to primitive head -- [1]
```

If you've defined concat the same way I did, then it will try to apply head to the integer 1, leading to an error. To solve this problem, we'll need to write a 'more recursive' function flatten that can deal with arbitrary nesting. This function will need to distinguish between integers and lists, so I've provided a primitive integer that tests whether its argument is an integer, returning a Boolean value:

```
>>> integer(2);;
--> true
>>> integer(nil);;
--> false
```

```
>>> integer(3 : nil);;
--> false
```

It's possible to write a version of flatten that assumes its argument is a list, but it's simpler to write a more general version that can be applied to single integers too, so that flatten(6) = [6]. The simpler, more general version will make your task easier in the rest of the lab, when you come to transform your program to imperative form. Implement flatten and test it.

Now we are going to implement a function flatsum that will compute the sum of all the integers that appear in a nested list. Thus flatsum(x) will be equivalent to sum(flatten(x)), but we aim to avoid first computing flatten(x) as a list. It should be easy to modify your definition of flatten to make a recursive definition of flatsum. Do it now and test the result.

1.3 Transforming into imperative form

The program for flatsum that you have written will be recursive: in fact, if you have taken the same approach as me, it will contain two recursive calls. Our aim now is to remove that recursion and make an imperative version of flatsum that uses a while loop instead of recursion.

The step from recursion to iteration is based on the observation that a recursive program where all the recursive calls occur at the end is equivalent to an imperative program with a loop. For example, the sum function,

```
val sum(xs) =
  let rec loop(ys, s) =
    if ys = nil then s else loop(tail(ys), s + head(ys)) in
  loop(xs, 0);;
```

can be rewritten as an imperative program like this:

```
val sumimp(xs) =
  let val ys = new() in let val s = new() in
  ys := xs; s := 0;
  while !ys <> nil do
    (s := !s + head(!ys); ys := tail(!ys));
  !s
```

The two parameters, ys and s, to the recursive function loop have become variables in the imperative program. Their initial values come from the initial call of loop, and the assignments that update them in the loop body come from the arguments that are passed to the recursive call of loop.

To perform a similar transformation on flatsum, we must first put it into tail-recursive form, where all the recursive calls occur 'at the end'. For example, if we had defined sum as

```
rec sum(xs) =
  if xs = nil then 0 else head(xs) + sum(tail(xs));;
```

then we couldn't have translated it immediately into a loop, because the recursive call sum(tail(xs)) is not the last thing to be done in the else branch. There is an addition to do after the recursive call has returned, and there is nowhere to put the addition when we try to write it as a loop. It's neccessary to rewrite sum in tail-recursive form before we can transform it into a loop, and we must now do the same to flatsum.

Your task at this point is to find a definition of flatsum that uses only tail recursion. As with sum, it will be necessary to introduce an auxiliary function with additional parameters; it may be helpful to think of one of these parameters as a stack of nested lists waiting to be summed. Call the new function flatsum1, write the definition and test it now.

Once you have a definition of flatsum in tail-recursive form, it should be straight-forward to rewrite the definition as a loop, without using any recursion. Call it flatsum2, write the definition and test it now.

1.4 Lab report

For the lab report, it will be sufficient to submit a listing of your file script.fun, with suitable comments about how you tested your solutions.

Chapter 2

Lab two: Memory and call-by-name

The combination of assignable variables and parameters passed by name is characteristic of the influential programming language Algol 60. That language, like all of the vast Algol family, had references as denoted but not expressed values, and had implicit dereferencing. We shall follow the pattern set by ML and have references as expressed values and explicit dereferencing, but many of the characteristic idioms of Algol 60's call-by-name will still be expressible in our language.

The exercise begins with a direct-style monadic interpreter for a language with assignable variables. For simplicity, the entire interpreter is provided in a separate directory from lab 1, though many of the files are similar and some (marked * in the list below) are identical.

Parsing.hs	Parsing library*
Environment.hs	Implementation of environments*
Memory.hs	Implementation of memories*
FunSyntax.hs	Abstract syntax of *Fun*
FunParser.hs	Parser for *Fun*
FunLab2.hs	Interpreter
jensen.fun	Example showing Jensen's device

All the changes you will make are in the file FunLab2.hs. As with Lab 1, a single command

```
% ghc --make FunLab2.hs -main-is FunLab2 -o fun
```

compiles the Haskell source into an executable interpreter program called *fun*.

The interpreter is based on the monad of memory, with memories storing values:

type $M\ \alpha = Mem \rightarrow (\alpha, Mem)$

type *Mem* = *Memory Value*

Provision has already been made for names that denote unevaluated parameters, because names are bound in the environment to objects of type *Def*, defined as follows.

data *Value* =

```
    Integer Integer
  | Bool Bool
  | Addr Location
  | Nil | Cons Value Value
  | Function ([Value] → M Value)

data Def =
    Const Value
  | Param (M Value)

type Env = Environment Def
```

Thus, in our language, *Value* is the domain of expressed values, and *Def* is the domain of denoted values. Remember that the type *M Value* can be thought of as a computational process that will produce a result of type *Value* but has not yet been executed; thus a *Def* object *Param xm* represents an argument that was not evaluated at the time of the procedure call. In the intepreter as it stands, however, these objects are never created, and parameters are all passed by value.

2.1 Passing parameters by name

Now we will change the semantics of procedures so that parameters are passed by name. The changes that are needed all flow from changing the type of function objects as follows.

```
data Value = ... | Function ([M Value] → M Value)
```

This new kind of function expects that the evaluation of its arguments will not have been executed before the function is called. Making this change will create a need for changes elsewhere, wherever function objects are created or used. The type of *apply* will change to

```
apply :: Value → [M Value] → M Value,
```

and the Haskell type-checker will highlight other places where changes are needed: in *abstract* (which should now use *Param* in place of *Const*), in the *eval* rule for *Apply* (where you will find that *evalargs* is no longer needed), and in the function *primitive* that creates primitives for the initial environment. All the primitives expect their arguments to have been evaluated already, and you can allow for this adjusting *primitive* to use a function

```
values :: [M α] → M [α]
```

that you will need to define. This function satisfies the specification

```
values (map (λ e → eval e env) es) = evalargs es env,
```

thus converting call-by-name into call-by-value.

2.2 Exploring call-by-name

Now that you have an interpreter that implements call-by-name, you can experiment with the feature. We can use assignable variables to show that parameters are evaluated at the time they are used:

```
% ./fun
>>> val x = new();;
--- x = <address 0>
>>> x := 3;;
--> 3
>>> val f(y) = x := 4; y;;
--- f = <function>
>>> f(!x);;
--> 4
```

Can you devise simple experiments to show that actual parameters are evaluated afresh each time they are used, and that if they are not used then they are not evaluated at all? ◁

These properties of parameters passed by name can be exploited in a famous programming trick called ‘Jensen’s device’, once popular in Algol 60. This uses a parameter passed by name to denote an expression that can be evaluated at different arguments. For example, consider the following program:

```
val sum(i, a, b, f) =
  let val s = new() in
  i := a; s := 0;
  while !i < b do (s := !s + f; i := !i + 1);
  !s;;

val go() =
  let val i = new() in
  sum(i, 0, 10, !i * !i);;
```

In sum(i, a, b, f), the parameter i must be a reference. The function sets i to each value in the interval $[a\,..\,b)$, and sums the values of f for these values of i. Under call-by-value, this program is pointless and just computes $(b - a) \times f$; but under call-by-name, the value of i can influence the value obtained by evaluating f, so the result is $\sum_{a \le i < b} f$. Thus the function go() computes $0^2 + 1^2 + 2^2 + \ldots + 9^2 = 285$.

This code is provided in the file jensen.fun, and you should check that it works with your interpreter. How would the same effect be achieved in a language which (like *Fun* before our modifications) has lambda expressions and call-by-value? Does the same solution work in the modified *Fun*? ◁

This is the end of the mandatory part of the lab; if you are continuing to the optional part that follows, it would be a good idea to save a copy of your file FunLab2.hs at this point, in case you make a mess of things later.

2.3 Explicit representation of name parameters (optional)

We have been representing parameters passed by name using the monad type *M Value*, which is shorthand for *Mem* $\rightarrow$ (*Value*, *Mem*). This means that our interpreter involves higher-order functions. We can move towards a concrete implementation by introducing an explicit representation for these unexecuted parameter evaluations. It would make sense to do this at the same time as introducing an explicit representation (closures) for function objects, but it is also possible to make this change independently, and that is what I suggest you do now.

If you study your interpreter, you will find that the objects that represent parameters are created by the expression *eval e env*, where *e* is an actual parameter expression, and *env* is the environment of the procedure call. For an explicit representation, we should therefore save *e* and *env* in a record, to be passed to *eval* when the value is needed. The traditional name for this closure-like object is a *thunk*, and we therefore define

data *Thunk* = *Thunk Expr Env*

and use this type wherever name parameters are used:

data *Value* = ... | *Function* ([*Thunk*] → *M Value*)

data *Def* = ... | *Param Thunk*

apply :: *Value* → [*Thunk*] → *M Value*

values :: [*Thunk*] → *M* [*Value*]

Naturally, the definitions of these functions and the places they are called must be updated to reflect the change.

After making these changes, the program in `jensen.fun` should run as it did before. If you like, you can also introduce closures for function values, following the pattern we used in the interpreter Fun_1: the result would be an interpreter for a language with memory, higher-order functions and call-by-name that does not itself depend essentially on any of these features in the metalanguage.

2.4 Lab report

As your lab report, you should submit a printed copy of your file `FunLab2.hs`, together with comments on how you tested your work, and answers to the questions in these instructions that are highlighted with '◁' in the margin.

Chapter 3

Lab three: Loops with exit

The programming language Oberon provides a kind of loop construct with embedded exit statements. For example, if initially $x = 0$, then the program

```
LOOP
  x := x+2;
  IF x > 3 THEN EXIT END;
  x := x+3
END
```

sets x to 7, because the whole loop body is executed once, setting x to 5, and on the second iteration x is increased from 5 to 7 before the exit statement is reached, terminating the loop.

In this lab, we will explore the meaning of these constructs. For simplicity, we will embed the new constructs into our language Fungol, extending the abstract syntax of expressions with these and other constructs:

```
data Expr =                      — Expressions e
    ...
  | Assign Expr Expr             — e₁ := e₂
  | Sequence Expr Expr           — e₁; e₂
  | Loop Expr                    — loop e₁
  | Exit                         — exit
```

As usual, parentheses can be used for grouping, so that the Oberon program shown above can be written as

```
loop (
  x := x+2;
  if x > 3 then exit else nil;
  x := x+3
)
```

In Fungol, every if must have an else, but nil provides a useful 'do nothing' expression to fill the gap. We'll also use nil as the value yielded by the loop expression when an exit has been reached, just as it is used as the value yielded by a terminating while loop in Fungol as it stands.

In Oberon, each exit statement must be embedded inside a loop statement, and is easy to see what is meant by the rule that "each exit statement terminates the *smallest textually enclosing* loop statement." Fungol opens up new possibilities, because it does not have Oberon's syntactic restriction that statements may not contain embedded declarations. Consider, for example, the following programs:

```
let rec f() = exit in loop f()

loop (let rec f() = exit in f())

loop (let rec f() = exit in loop f())
```

In the first program, the exit statement is not *textually* inside any loop, though a loop is active at the time the function f is called; it seems that this program should be illegal. In the second program, both the definition of f and the invocation of it are inside the same loop, and it seems that this must be the loop that terminates when the exit statement is executed. In the third program, the definition of f is inside one loop, but its invocation is inside another loop nested within it. Using the textual rule, it seems that the outer loop must be the one that is terminated by the exit statement.

Although this rule sounds static, it means that in a language that allows function definitions to be nested inside loops, defining a function entails capturing the dynamic context of the definition, so that exit statements within the function body can terminate the appropriate loop. This capture of the context can be neatly modelled using continuations, as we shall see. Of course, Oberon does not allow loops and functions to be nested in this way, and we might argue that our semantics reveals why this was a wise choice, because it underlines the fact that the more restricted constructs of Oberon are significantly easier to implement.

3.1 Getting started

The lab materials contain an exact copy of the Fungol interpreter discussed in lectures, and we are going to use that as the basis for an interpreter that also implements the loop and exit constructs. The first step is to extend the abstract syntax and the parser for Fungol to handle the loop and exit constructs.

The lab materials consist of these files:

FunSyntax.hs	Abstract syntax of Fun
FunParser.hs	Parser for Fun
Fungol.hs	Interpreter for Fungol dialect
Environment.hs	Usual implementation of environments
Memory.hs	Usual implementation of memory
Parsing.hs	Parsing library
Cont.hs	Useful code fragments for continuations (see Figure 3.1)

You can build the interpreter with the command,

```
% ghc --make Fungol.hs -main-is Fungol -o fungol
```

As usual, the complete implementation consists of several modules, in addition to the interpreter itself, and you will need to modify several of them, in order to add the syntax for the new constructs.

- In the *FunSyntax* module, you will need to add abstract syntax for the loop and exit constructs.
- In the *FunParser* module, you will need to add to the type *token* that is returned by the lexer, and extend the lexer's table of keywords by adding loop and exit.
- Then you can extend the parser (in the same module) to recognize the forms loop e_1 and exit as expressions. Look for the syntax of the existing while loops to find where to add the new kind of loop, and add exit in *p_primary*. The parser is written in a monadic style that we may have time to look at later in the course; for now, you will probably be able to make progress by copying fragments of the existing parser.

Once these preliminary changes are complete, it should be possible to parse functions containing loop statements, such as the file ex0.fun that's provided with lab materials. Attempting to call the function loop0 that's defined in that file will (of course) result in an interpreter crash.

3.2 First model

The next step is to implement loop and exit in the interpreter. A simple way of doing this (adequate for Oberon but not for Fungol) is provided by Exercise 3.5, where the usual 'monad of memory' is replaced by the type

$$\textbf{type}\ M\ \alpha = \mathit{Mem} \to (\mathit{Maybe}\ \alpha, \mathit{Mem})$$

This type supports the usual monad operations, in addition to operations connected with memory and with abnormal termination:

$$\begin{aligned}&\mathit{new} :: M\ \mathit{Location}\\ &\mathit{get} :: \mathit{Location} \to M\ \mathit{Value}\\ &\mathit{put} :: \mathit{Location} \to \mathit{Value} \to M\ ()\end{aligned}$$

$$\begin{aligned}&\mathit{exit} :: M\ \alpha\\ &\mathit{orelse} :: M\ \alpha \to M\ \alpha \to M\ \alpha\end{aligned}$$

The idea is that (rather like what happens with exceptions) an expression can either yield a value in the usual way, or it can invoke exit and lead to termination of any enclosing loop construct. Let's agree to give the name *exit* to the operation that was formerly known as *failure*, since exiting from a potentially infinite loop doesn't seem like a failure!

You should use this monad as the basis for a first implementation of loop and exit. To do this, you need to modify Fungol.hs as follows:

- Replace the monad type M, and adjust the definitions of *result*, $\triangleright$, *get*, *put* and *new* to suit.
- Add new operations *exit* :: $M\ \alpha$ and *orelse* :: $M\ \alpha \to M\ \alpha \to M\ \alpha$ as discussed above.
- Leaving the existing clauses for *eval* untouched, add new clauses that implement the forms loop e_1 and exit. For the loop construct, you will need to combine a recursive definition that reflects the looping aspect of its behaviour with an application of *orelse* that reflects the exit-capturing aspect.

- Adjust the top-level function *obey* so that it fits in with the changed type *M* and prints a message for programs where an exit statement occurs outside a loop.

Now it should be possible to run the program in file ex0.fun, as well as others of your own devising. You might like to verify that an exit statement inside nested loops exits the inner loop and not the outer one.

In this part of the lab, it is advantageous to work tidily, so that (for example) the detail of what type is used in the monad is hidden inside operations like *get* and *orelse*. That will mean that the existing equations for *eval* will not need to change, and the new ones you introduce will also remain unchanged when we move to a continuation-based monad later.

3.3 Evaluating the model

In simple examples, you will find that your interpreter does what is expected. More complex programs can cause problems, however. Try to puzzle out what your interpreter will do for each of the three problem programs that were discussed in the introduction to the lab, then verify by experiment that your prediction is right. The effects seen here can be summarized by saying that the exit construct terminates the *dynamically closest* loop, and function calls make it possible that this loop is not the same as the one that textually encloses the exit, or indeed that an exit that terminates a loop does not appear inside a loop at all. Allowing function calls in the language (and allowing functions to be nested inside loops) therefore makes our interpreter a poor model for a language where exit constructs are associated with their textually enclosing loop.

3.4 A model based on continuations

We've seen that allowing function definitions to be nested inside loops creates a problem with determining which loop should terminate if the body of a function contains an exit statement. One possibility – the one that follows naturally from our *Maybe*-based model – is that the loop that terminates is the one that is *dynamically* innermost; but we might also try to work out whether we can implement a rule that says the loop terminates is the smallest one that *statically* or *textually* contains the exit statement. For this, an alternative model of the semantics based on continuations will be useful.

All this discussion of static vs. dynamic association of exit statements with their containing loops become moot in a language which (like Oberon) does not permit function definitions to be nested inside loops at all, and forbids exit statements that are not textually embedded in a containing loop statement. Perhaps the complexity of the semantic choices to be made (together with the ease of implementing the simpler scheme) gives a good explanation of why Oberon and other languages have adopted these rules.

For the continuation-based model, we define as usual a continuation to be a function that captures the 'future' of a computation. It expects to receive a value (of some type α) and a memory state, and it delivers the final answer from the whole program.

$$\textbf{type}\ \mathit{Cont}\ \alpha = \alpha \to \mathit{Mem} \to \mathit{Answer}$$

Now we are going to define a monad where each computation receives not one but two continuations. One continuation, of type *Cont* ( ), is invoked if the computation ends with an exit statement, and the other, of type *Cont* α is invoked if the computation produces an ordinary result of type α.

$$\textbf{type}\ M\ \alpha = \mathit{Mem} \to \mathit{Cont}\ () \to \mathit{Cont}\ \alpha \to \mathit{Answer}$$

We'll use the name *kx* for an eXit continuation of type *Cont* ( ), and *ks* for a Success continuation of type *Cont* α for some type α. In detail, if a computation r terminates normally when started with memory state *mem*, then $r\ \mathit{mem}\ \mathit{kx}\ \mathit{ks} = \mathit{ks}\ x\ \mathit{mem}'$, where x is the value returned by r, and mem' is the modified memory state. If, on the other hand, r finishes by executing an exit statement, then $r\ \mathit{mem}\ \mathit{kx}\ \mathit{ks} = \mathit{kx}\ ()\ \mathit{mem}$.

The monad operations can be defined as follows:

$$
\begin{aligned}
&\mathit{result} :: \alpha \to M\ \alpha\\
&\mathit{result}\ x\ \mathit{mem}\ \mathit{kx}\ \mathit{ks} = \mathit{ks}\ x\ \mathit{mem}
\end{aligned}
$$

$$
\begin{aligned}
&(\triangleright) :: M\ \alpha \to (\alpha \to M\ \beta) \to M\ \beta\\
&(\mathit{xm} \triangleright f)\ \mathit{mem}\ \mathit{kx}\ \mathit{ks} =\\
&\quad \mathit{xm}\ \mathit{mem}\ \mathit{kx}\ (\lambda\ x\ \mathit{mem}' \to f\ x\ \mathit{mem}'\ \mathit{kx}\ \mathit{ks})
\end{aligned}
$$

So *result* simply invokes the success continuation, and $\triangleright$ performs the two steps in sequence, passing the same exit continuation to both of them. In $\mathit{xm} \triangleright f$, if the first step *xm* invokes its exit continuation, then the second step f won't be invoked at all, and this is what we want.

It's also possible to redefine the operations involving the memory to work with this monad: they continue to have types with the same shape:

$$
\begin{aligned}
&\mathit{get} :: \mathit{Location} \to M\ \mathit{Value}\\
&\mathit{put} :: \mathit{Location} \to \mathit{Value} \to M\ ()\\
&\mathit{new} :: M\ \mathit{Location}
\end{aligned}
$$

(but of course with a different M from before). The details are left for you to work out.

What's more, we can redefine the operations associated with exit and add a couple more:

$$\mathit{exit} :: M\ \alpha$$

$$\mathit{orelse} :: M\ \alpha \to M\ \alpha \to M\ \alpha$$

$$
\begin{aligned}
&\mathit{callxc} :: (\mathit{Cont}\ () \to M\ \alpha) \to M\ \alpha\\
&\mathit{callxc}\ f\ \mathit{mem}\ \mathit{kx}\ \mathit{ks} = f\ \mathit{kx}\ \mathit{mem}\ \mathit{kx}\ \mathit{ks}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{withxc} :: \mathit{Cont}\ () \to M\ \alpha \to M\ \alpha\\
&\mathit{withxc}\ \mathit{kx}\ \mathit{xm}\ \mathit{mem}\ \mathit{kx}'\ \mathit{ks} = \mathit{xm}\ \mathit{mem}\ \mathit{kx}\ \mathit{ks}
\end{aligned}
$$

The idea of these last two is that *callxc f* calls the function f, passing it a copy of the current exit continuation, and *withxc kx xm* is like the computation *xm*, except that *kx* is used in place of the current exit continuation.

By using *withxc*, it's possible to define a version of *abstract* that has the modified type,

$$\mathit{abstract} :: [\mathit{Ident}] \to \mathit{Expr} \to \mathit{Env} \to \mathit{Cont}\ () \to \mathit{Def}$$

so that *abstract xs e env kx* creates a procedure value that, when activated, evaluates the expression *e* in an environment derived from *env*, using *kx* as the exit continuation. Thus not only the environment *env* but also the exit continuation *kx* become statically bound. This version of *abstract* can in turn by used in a new equation defining the meaning of function definitions, where *callxc* can be used to capture the exit continuation at the point of definition.

To build an implementation for our extended Fungol based on this monad, you can use the same modified syntax and parser as before, but you will need to make a new interpreter with the following changes. (Keep a copy of your first version too.)

- Replace the monad type with the one based on continuations, and redefine the operations *result*, $\triangleright$, *get*, *put*, *new*, *exit* and *orelse* to work with the new type.
- Replace the *abstract* function in the interpreter with one that statically binds an exit continuation.
- Replace the defining equation for *elab* (*Rec x* (*Lambda* ...)) with one that captures the exit continuation.
- Add the type definition

  type *Answer* = (*String*, *GloState*)

  and adjust the top level function *obey* to fit with the new types for *eval* and *elab*.

To save you some copy-typing, the file Cont.hs that's provided with the lab materials and shown in Figure 3.1 contains some of the code you will need. After completing these changes, it should be possible to re-run the examples from Section 3.3 and get different results.

3.5 Lab report

Produce listings of your two interpreters as your lab report, and be ready to demonstrate and explain the results that are obtained when your interpreters run the example programs.

```
— Code for continuation-based model

type Cont a = a → Mem → Answer

type M a = Mem → Cont () → Cont a → Answer

result x mem kx ks = ks x mem

(xm ▷ f) mem kx ks =
  xm mem kx (λ x mem′ → f x mem′ kx ks)

—

callxc :: (Cont () → M a) → M a
callxc f mem kx ks = f kx mem kx ks

withxc :: Cont () → M a → M a
withxc kx xm mem kx′ ks = xm mem kx ks

—

type GloState = (Env, Mem)
type Answer = (String, GloState)

obey :: Para → GloState → (String, GloState)
obey (Calculate exp) (env, mem) =
  eval exp env mem
    (λ () mem′ → ("∗∗∗exit in main program∗∗∗", (env, mem′)))
    (λ v mem′ → (print_value v, (env, mem′)))
obey (Define def) (env, mem) =
  let x = def_lhs def in
  elab def env mem
    (λ () mem′ → ("∗∗∗exit in definition∗∗∗", (env, mem′)))
    (λ env′ mem′ → (print_defn x (find env′ x), (env′, mem′)))

main = dialog funParser obey (init_env, init_mem)
```

Figure 3.1: *Contents of file* Cont.hs