Implementing Functional Languages: a tutorial

Simon L Peyton Jones
Department of Computing Science, University of Glasgow

and David R Lester
Department of Computer Science, University of Manchester

March 12, 1997

This book is dedicated to the memory of our friend and colleague Jon Salkild (1963-1991).

Contents

Preface

This book gives a practical approach to understanding implementations of non-strict functional languages using lazy graph reduction. The book is intended to be a source of practical labwork material, to help make functional-language implementations 'come alive', by helping the reader to develop, modify and experiment with some non-trivial compilers.

The unusual aspect of the book is that it is meant to be *executed* as well as *read*. Rather than merely presenting an abstract description of each implementation technique, we present the code for a complete working prototype of each major method, and then work through a series of improvements to it. All of the code is available in machine-readable form.

Overview of the book

The principal content of the book is a series of implementations of a small functional language called the *Core language*. The Core language is designed to be as small as possible, so that it is easy to implement, but still rich enough to allow modern non-strict functional languages to be translated into it without losing efficiency. It is described in detail in Chapter 1, in which we also develop a parser and pretty-printer for the Core language.

Appendix B contains a selection of Core-language programs for use as test programs thoughout the book.

The main body of the book consists of four distinct implementations of the Core language.

- Chapter 2 describes the most direct implementation, based on *template instantiation*.
- Chapter 3 introduces the *G-machine*, and shows how to compile the program to sequences of instructions (G-code) which can be further translated to machine code.
- Chapter 4 repeats the same exercise for a different abstract machine, the *Three Instruction Machine* (TIM), whose evaluation model is very different from that of the G-machine. The TIM was developed more recently than the G-machine, so there is much less other literature about it. Chapter 4 therefore contains a rather more detailed development of the TIM's evaluation model than that given for the G-machine.
- Finally, Chapter 5 adds a new dimension by showing how to compile functional programs for a *parallel G-machine*.

For each of these implementations we discuss two main parts, the *compiler* and the *machine*

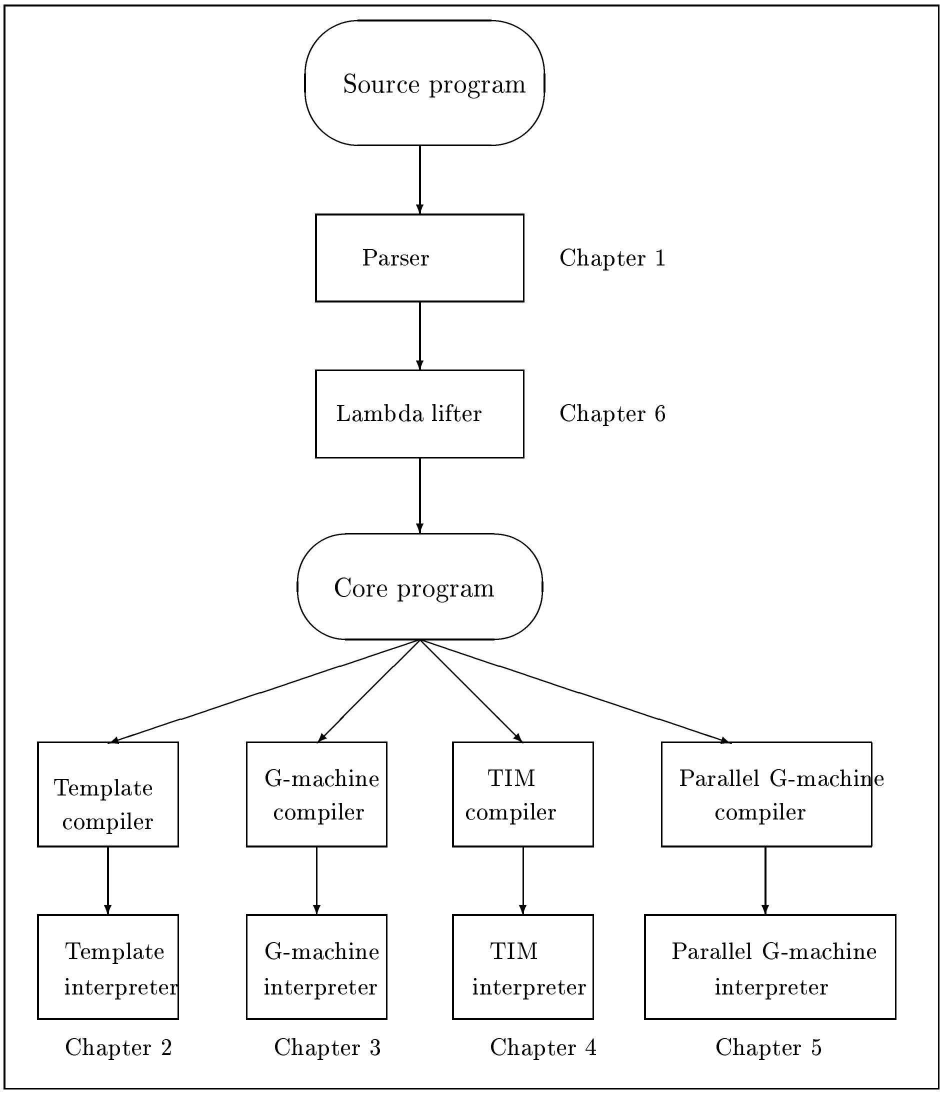


Figure 0.1: Overview of the implementations

interpreter. The compiler takes a Core-language program and translates it into a form suitable for execution by the machine interpreter.

The machine interpreter simulates the execution of the compiled program. In each case the interpreter is modelled as a *state transition system* so that there is a very clear connection between the machine interpreter and a 'real' implementation. Figure 0.1 summarises the structure of our implementations.

One important way in which the Core language is restrictive is in its lack of local function definitions. There is a well-known transformation, called *lambda lifting*, which turns local function definitions into global ones, thus enabling local function definitions to be written freely and transformed out later. In Chapter 6 we develop a suitable lambda lifter. This chapter is more than just a re-presentation of standard material. *Full laziness* is a property of functional programs which had previously been seen as inseparable from lambda lifting. In Chapter 6 we show that they are in fact quite distinct, and show how to implement full laziness in a separate pass from lambda lifting.

Throughout the book we use a number of utility functions and data types which are defined in Appendix A.

Some sections and exercises are a little more advanced, and can be omitted without major loss. They are identified with a dagger, thus: †.

The prototyping language

The question arises of what language to use for writing our implementations. We have chosen to use an existing functional language, Miranda[1]. One of the major uses of functional languages is for rapid prototyping, because they allow us to express the fundamental aspects of the prototype without getting bogged down in administrative detail. We hope that this book can serve as a large example to substantiate this claim. In addition, working through this book should provide useful experience of writing substantial functional programs.

This book is not an introduction to functional programming. We assume that you have done some programming in a functional language already. (Suitable introductions to functional programming include [Bird and Wadler 1988] and [Holyer 1991].) Nevertheless, the programs developed in this book are quite substantial, and we hope they will serve as a role model, to stretch and develop your ability to write clear, modular functional programs.

Miranda code is written in typewriter fount using the 'inverse comment convention'. For example, here is a definition of a function which takes the length of a list:

```
> length [] = 0
> length (x:xs) = 1 + length xs
```

The `>` mark in the left margin indicates that this is a line of executable Miranda code. Not only does this distinguish Miranda code from Core-language programs (which are also written in typewriter fount), but Miranda itself recognises this convention, so the text of each chapter of this book *is* executable Miranda code! Indeed, it has all been executed. (Actually, a small amount of pre-processing is required before feeding the text to Miranda, because we sometimes write several versions of the same function, as we refine the implementation, and Miranda objects to such multiple definitions. The pre-processing is simply a selection process to pick the particular version we are interested in.)

The text contains *all* the code required for the initial version of each implementation. Occasionally, this becomes rather tiresome, because we have to present chunks of code which are not very interesting. Such chunks could have been omitted from the printed text (though not from the executable version), but we have chosen to include them, so that you can always find a definition for every function. (The index contains an entry for every Miranda function definition.)

Like most functional languages, Miranda comes with a collection of pre-declared functions, which are automatically in scope. We make use of these throughout the text, referring to them as *standard functions*. You can find details of all the standard functions in Miranda's online manual.

[1]Miranda is a trade mark of Research Software Ltd.

What this book does not cover

We focus exclusively in this book on the 'back end' of functional-language compilers. We make no attempt to discuss how to translate programs written in a fully fledged functional language, such as Miranda, into the Core language, or how to type-check such programs.

The development throughout is informal. It would be nice to give a formal proof of the equivalence between the meaning of a Core program and its implementation, but this is quite a hard task. Indeed, the only full proof of such an equivalence which we know of is [Lester 1988].

Relationship to *The implementation of functional programming languages*

An earlier book by one of us, [Peyton Jones 1987], covers similar material to this one, but in a less practically oriented style. Our intention is that a student should be able to follow a course on functional-language implementations using the present book alone, without reference to the other.

The scope of this book is somewhat more modest, corresponding to Parts 2 and 3 of [Peyton Jones 1987]. Part 1 of the latter, which discusses how a high-level functional language can be translated into a core language, is not covered here at all.

Getting the machine-readable sources

You can get all the source code for this book, by network file transfer (FTP) from the several sites. In each case, you need only get the file

```
pjlester-n.m.tar.Z
```

where **n.m** is the current version number of the book. There is always only one such file, but the **n.m** may vary as we correct errors and otherwise improve the material. Once you have got the file, run the command

```
zcat pjlester-n.m.tar.Z | tar xf -
```

and then read or print the `README` file, and the DVI file `installation.dvi`. If you don't have `zcat`, do the following instead:

```
uncompress pjlester-n.m.tar.Z
tar xf pjlester-n.m.tar
```

The sites where the sources are held are as follows.

Site	Host name	Host address	Directory
Glasgow	`ftp.dcs.glasgow.ac.uk`	130.209.240.50	`pub/pj-lester-book`
Yale	`nebula.cs.yale.edu`	128.36.13.1	`pub/pj-lester-book`
Chalmers	`animal.cs.chalmers.se`	129.16.2.27	`pub/pj-lester-book`

Log in as `anonymous`, and use your electronic mail address as password. Here is a sample Internet FTP session:

```
% ftp nebula.cs.yale.edu
Connected to NEBULA.SYSTEMSZ.CS.YALE.EDU.
220 nebula FTP server (SunOS 4.0) ready.
Name (nebula.cs.yale.edu:guestftp): anonymous
331 Guest login ok, send ident as password.
Password: simonpj@dcs.glasgow.ac.uk
230 Guest login ok, access restrictions apply.
ftp> type binary
200 Type set to I.
ftp> cd pub/pj-lester-book
250 CWD command successful.
ftp> get pjlester-1.2.tar.Z
<messages about a successful transfer would come here>
ftp> bye
```

Within the UK, you may get the above file from `uk.ac.glasgow.dcs` by anonymous UK NIFTP (binary mode; user: `guest`; password: your e-mail address); request:

`<FP>/pj-lester-book/`**filename**

A typical command you might use on at least some Unix machines is:

```
cpf -b '<FP>/pj-lester-book/pjlester-1.2.tar.Z@uk.ac.glasgow.dcs'
       pjlester-1.2.tar.Z
```

Errors

We would greatly appreciate your help in eliminating mistakes in the text. If you uncover any errors, please contact one of us at the addresses given below.

Acknowledgements

We would like to thank Professor Rajaraman from the Indian Institute of Science in Bangalore for the invitation to give the course on which this book is based, and the British Council for sponsoring the visit.

A number of people have made helpful comments on drafts of this material. We thank them very much, especially Guy Argo, Cordelia Hall, Denis Howe, John Keane, Nick North, David Wakeling and an anonymous reviewer.

Simon L. Peyton Jones
Department of Computing Science
University of Glasgow G12 8QQ
email: `simonpj@dcs.glasgow.ac.uk`

David R. Lester
Department of Computer Science
University of Manchester M13 9PL
email: `dlester@cs.manchester.ac.uk`

Chapter 1

The Core language

All our implementations take some program written in a simple *Core language* and execute it. The Core language is quite impoverished, and one would not want to write a large program in it. Nevertheless it has been carefully chosen so that it is possible to translate programs in a rich functional language (such as Miranda) into the Core language without losing expressiveness or efficiency. The Core language thus serves as a clean interface between the 'front end' of the compiler, which is concerned with high-level language constructs, and the 'back end', which is concerned with implementing the Core language in various different ways.

We begin with an informal introduction to the Core language (Section 1.1). Following this, we define the Core language more formally, by giving:

- Its syntax (Section 1.2).
- Miranda data types `coreProgram` and `coreExpr` for Core-language programs and expressions respectively (Section 1.3). These will serve as the input data types for the compilers we build.
- Definitions for a small *standard prelude* of Core-language functions, which will be made available in any Core program (Section 1.4).
- A pretty-printer, which transforms a Core-language program into a character string which, when printed, is a formatted version of the program (Section 1.5).
- A parser, which parses a character string to produce a Core-language program (Section 1.6).

This chapter has a second purpose: it introduces and uses many of the features of Miranda which we will use throughout the book before we get involved with any of our functional-language implementations.

1.1 An overview of the Core language

Here is an example Core program[1], which evaluates to 42:

```
main = double 21
double x = x + x
```

A Core program consists of a set of *supercombinator definitions*, including a distinguished one, `main`. To execute the program, we evaluate `main`. Supercombinators can define functions, such as the definition of `double`. `double` is a function of one argument, `x`, which returns twice its argument. The program looks quite similar to the top level of a Miranda script, except that no pattern matching is permitted for function arguments. Pattern matching is performed by a separate Core language construct, the `case` expression, which is discussed below. Each supercombinator is defined by a single equation whose arguments are all simple variables.

Notice that not all supercombinators have arguments. Some, such as `main`, take no arguments. Supercombinators with no arguments are also called *constant applicative forms* or CAFs and, as we shall see, often require special treatment in an implementation.

1.1.1 Local definitions

Supercombinators can have local definitions, using the `let` construct of the Core language:

```
main = quadruple 20 ;
quadruple x = let twice_x = x+x
              in twice_x + twice_x
```

Here `twice_x` is defined locally within the body of `quadruple` to be `x+x`, and `quadruple` returns `twice_x + twice_x`. Like Miranda `where` clauses, local definitions are useful both to name intermediate values, and to save recomputing the same value twice; the programmer can reasonably hope that only two additions are performed by `quadruple`.

A `let` expression is *non-recursive*. For recursive definitions, the Core language uses the `letrec` construct, which is exactly like `let` except that its definitions can be recursive. For example:

```
infinite n = letrec ns = cons n ns
             in ns
```

The reason that we distinguish `let` from `letrec` in the Core language (rather than providing only `letrec`) is that `let` is a bit simpler to implement than `letrec`, and we may get slightly better code.

`let` and `letrec` are similar to the Miranda `where` clause, but there are a number of important differences:

- The `where` clause always defines a recursive scope. There is no non-recursive form.

[1] We use typewriter fount for Core programs, but without the initial > sign which distinguishes executable Miranda code.

- A `where` clause can be used to define local functions, and to perform pattern matching:

```
... where f x = x+y
          (p,q) = zip xs ys
```

  Neither of these facilities is provided by the Core language `let` and `letrec` expressions. Functions can only be defined at the top level, as supercombinators, and pattern matching is done only by `case` expressions.

  In short, *the left-hand side of a `let` or `letrec` binding must be a simple variable.*

- The `let`/`letrec` construct is an *expression*. It is therefore quite legal to write (for example):

```
quad_plus_one x = 1 + (let tx = x+x in tx+tx)
```

  In contrast a `where` clause in Miranda can only be attached to *definitions*. (One reason for this is that it allows the definitions in a Miranda `where` clause to range over several guarded right-hand sides.)

1.1.2 Lambda abstractions

Functions are usually expressed in the Core language using top-level supercombinator definitions, and for most of the book this is the *only* way in which functions can be denoted. However, it is sometimes convenient to be able to denote functions using explicit *lambda abstractions*, and the Core language provides a construct to do so. For example, in the program

```
double_list xs = map (\ x. 2*x) xs
```

the lambda abstraction `(\ x. 2*x)` denotes the function which doubles its argument.

It is possible to transform a program involving explicit lambda abstractions into an equivalent one which uses only top-level supercombinator definitions. This process is called *lambda lifting*, and is discussed in detail in Chapter 6. Throughout the other chapters we assume that this lambda lifting process has been done, so they make no use of explicit lambda abstractions.

The final major construct in the Core language is the `case` expression, which expresses pattern matching. There are several ways of handling pattern matching, so we begin with a review of structured data types.

1.1.3 Structured data types

A universal feature of all modern functional programming languages is the provision of *structured types*, often called *algebraic data types*. For example, here are a few algebraic type definitions, written in Miranda:

```
colour ::= Red | Green | Blue
```

```
complex ::= Rect num num | Polar num num

numPair ::= MkNumPair num num

tree * ::= Leaf * | Branch (tree *) (tree *)
```

Each definition introduces a new *type* (such as `colour`), together with one or more *constructors* (such as `Red`, `Green`). They can be read as follows: 'A value of type `colour` is either `Red` or `Green` or `Blue`', and 'A `complex` is either a `Rect` containing two `num`s, or a `Polar` containing two `num`s'.

The type `tree` is an example of a *parameterised* algebraic data type; the type `tree` is parameterised with respect to the type variable `*`. It should be read as follows: 'a `tree` of `*`'s is either a `Leaf` containing a `*`, or a `Branch` containing two `tree` of `*`'s'. Any particular tree must have leaves of uniform type; for example, the type `tree num` is a tree with `num`s at its leaves, and the type `tree colour` is a tree with `colour`s at its leaves.

Structured values are *built* with these constructors; for example the following expressions denote structured values:

```
Green
Rect 3 4
Branch (Leaf num) (Leaf num)
```

Structured values are *taken apart* using *pattern matching*. For example:

```
isRed Red   = True
isRed Green = False
isRed Blue  = False

first (MkNumPair n1 n2) = n1

depth (Leaf n)       = 0
depth (Branch t1 t2) = 1 + max (depth t1) (depth t2)
```

Several data types usually thought of as 'built in' are just special cases of structured types. For example, booleans are a structured type: they can be defined by the algebraic data type declaration

```
bool ::= False | True
```

Apart from their special syntax, the lists and tuples provided by Miranda are further examples of structured types. If we use `Cons` and `Nil` as constructors rather than the special syntax of `:` and `[]`, we could define lists like this:

```
list * ::= Nil | Cons * (list *)
```

Chapter 4 of [Peyton Jones 1987] gives a fuller discussion of structured types.

The question arises, therefore: *how are we to represent and manipulate structured types in our small Core language?* In particular, our goal is to avoid having data type declarations in the Core language altogether. The approach we take breaks into two parts:

- Use a simple, uniform representation for constructors.
- Transform pattern matching into simple `case` expressions.

1.1.4 Representing constructors

Instead of allowing user-defined constructors such as `Red` and `Branch` in our Core language, we provide a *single* family of constructors

`Pack{`**tag**, **arity**`}`

Here, **tag** is an integer which uniquely identifies the constructor, and **arity** tells how many arguments it takes. For example, we could represent the constructors of `colour`, `complex`, `tree` and `numPair` as follows:

```
Red        =  Pack{1,0}
Green      =  Pack{2,0}
Blue       =  Pack{3,0}

Rect       =  Pack{4,2}
Polar      =  Pack{5,2}

Leaf       =  Pack{6,1}
Branch     =  Pack{7,2}

MkNumPair  =  Pack{8,2}
```

So in the Core language one writes

```
Pack{7,2} (Pack{6,1} 3) (Pack{6,1} 4)
```

instead of

```
Branch (Leaf 3) (Leaf 4)
```

The tag is required so that objects built with different constructors can be distinguished from one another. In a well-typed program, objects of different type will never need to be distinguished at run-time, so tags only need to be unique *within a data type.* Hence, we can start the tag at 1 afresh for each new data type, giving the following representation:

```
Red        =  Pack{1,0}
Green      =  Pack{2,0}
Blue       =  Pack{3,0}

Rect       =  Pack{1,2}
Polar      =  Pack{2,2}

Leaf       =  Pack{1,1}
Branch     =  Pack{2,2}

MkNumPair  =  Pack{1,2}
```

1.1.5 `case` expressions

In general, the pattern matching allowed by modern functional programming languages can be rather complex, with multiple nested patterns, overlapping patterns, guards and so on. For the Core language, we eliminate these complications by outlawing all complex forms of pattern matching! We do this by providing only `case` *expressions* in the Core language. Their formal syntax is given in Section 1.2, but here are some examples:

```
isRed c = case c of
                <1> -> True ;
                <2> -> False ;
                <3> -> False

depth t = case t of
                <1> n -> 0 ;
                <2> t1 t2 -> 1 + max (depth t1) (depth t2)
```

The important thing about `case` expressions is that each alternative consists only of a tag followed by a number of variables (which should be the same as the arity of the constructor). No nested patterns are allowed.

`case` expressions have a very simple operational interpretation, rather like a multi-way jump: evaluate the expression to be analysed, get the tag of the constructor it is built with and evaluate the appropriate alternative.

1.2 Syntax of the Core language

Figure 1.1 gives the syntax for the Core language. The grammar allows infix binary operators, but (for brevity) is not explicit about their precedence. Instead we give the following table of precedences, where a higher precedence means tighter binding:

Precedence	Associativity	Operator
6	Left	Application
5	Right	`*`
	None	`/`
4	Right	`+`
	None	`-`
3	None	`== ~= > >= < <=`
2	Right	`&`
1	Right	`|`

An operator's associativity determines when parentheses may be omitted around repetitions of the operator. For example, `+` is right-associative, so `x+y+z` means the same as `x+(y+z)`. On the other hand, `/` is non-associative, so the expression `x/y/z` is illegal.

There is no special operator symbol for unary negation. Instead, the `negate` function is provided, which behaves syntactically like any normal function. For example:

```
f x = x + (negate x)
```

The boolean negation operator, `not`, is handled in the same way.

1.3 Data types for the Core language

For each of the implementations discussed in this book we will build a compiler and a machine interpreter. The compiler takes a Core program and translates it into a form suitable for execution by the machine interpreter. To do this we need a Miranda data type to represent Core programs, and that is what we will define in this section. In fact we will define a type for Core programs, one for Core expressions and a few other auxiliary types.

The data type of Core-language expression, `expr`, is defined as follows:

```
> expr *
>   ::=  EVar name                      || Variables
>        | ENum num                     || Numbers
>        | EConstr num num              || Constructor tag arity
>        | EAp (expr *) (expr *)        || Applications
>        | ELet                         || Let(rec) expressions
>             isRec                     ||   boolean with True = recursive,
>             [(*, expr *)]             ||   Definitions
>             (expr *)                  ||   Body of let(rec)
>        | ECase                        || Case expression
>             (expr *)                  ||   Expression to scrutinise
>             [alter *]                 ||   Alternatives
>        | ELam [*] (expr *)            || Lambda abstractions
```

We choose to parameterise the data type of `expr` with respect to its *binders*. A binder is the name used at the binding occurrence of a variable; that is, on the left-hand side of a `let(rec)`

Programs	**program**	$\rightarrow$	**sc**$_1$; … ; **sc**$_n$	$n \geq 1$
Supercombinators	**sc**	$\rightarrow$	**var var**$_1$ … **var**$_n$ = **expr**	$n \geq 0$
Expressions	**expr**	$\rightarrow$	**expr aexpr**	Application
		$\mid$	**expr**$_1$ **binop expr**$_2$	Infix binary application
		$\mid$	`let` **defns** `in` **expr**	Local definitions
		$\mid$	`letrec` **defns** `in` **expr**	Local recursive definitions
		$\mid$	`case` **expr** `of` **alts**	Case expression
		$\mid$	`\` **var**$_1$ … **var**$_n$ `.` **expr**	Lambda abstraction ($n \geq 1$)
		$\mid$	**aexpr**	Atomic expression
	aexpr	$\rightarrow$	**var**	Variable
		$\mid$	**num**	Number
		$\mid$	`Pack{`**num**`,`**num**`}`	Constructor
		$\mid$	`(` **expr** `)`	Parenthesised expression
Definitions	**defns**	$\rightarrow$	**defn**$_1$; … ; **defn**$_n$	$n \geq 1$
	defn	$\rightarrow$	**var** = **expr**	
Alternatives	**alts**	$\rightarrow$	**alt**$_1$; … ; **alt**$_n$	$n \geq 1$
	alt	$\rightarrow$	`<`**num**`>` **var**$_1$ … **var**$_n$ `->` **expr**	$n \geq 0$
Binary operators	**binop**	$\rightarrow$	**arithop** $\mid$ **relop** $\mid$ **boolop**	
	arithop	$\rightarrow$	`+` $\mid$ `-` $\mid$ `*` $\mid$ `/`	Arithmetic
	relop	$\rightarrow$	`<` $\mid$ `<=` $\mid$ `==` $\mid$ `~=` $\mid$ `>=` $\mid$ `>`	Comparison
	boolop	$\rightarrow$	`&` $\mid$ $\mathtt{\vert}$	Boolean
Variables	**var**	$\rightarrow$	**alpha varch**$_1$ … **varch**$_n$	$n \geq 0$
	alpha	$\rightarrow$	*an alphabetic character*	
	varch	$\rightarrow$	**alpha** $\mid$ **digit** $\mid$ `_`	
Numbers	**num**	$\rightarrow$	**digit**$_1$ … **digit**$_n$	$n \geq 1$

Figure 1.1: BNF syntax for the Core language

definition, or in a lambda abstraction. The declaration can be read 'An `expr` of `*` is either an `EVar` containing a `name`, or ..., or an `ELam` containing a list of values of type `*` and an `expr` of `*`'.

For the most of the book we always use `name` in these binding positions, so we use a *type synonym* to define the type of `coreExpr`, which is the type we will normally use:

```
> coreExpr == expr name
```

The ability to use types other than `name` in binding positions is only used in Chapter 6.

Apart from this, the data type follows fairly directly from the syntax given in the previous section, except that various superficial differences are discarded. The biggest difference is that infix operators are expressed in prefix form in the data type. For example, the expression

```
        x + y
```

is represented by

```
        EAp (EAp (EVar "+") (EVar "x")) (EVar "y")
```

Variables are represented by an `EVar` constructor containing the variable's name. A variable's name is represented simply by a list of characters, which we express using another type synonym:

```
> name == [char]
```

Constructors are identified by their arity and tag, as described in Section 1.1.4.

`let` and `letrec` expressions are represented by an `ELet` constructor containing: a flag of type `isRec` to distinguish the recursive case from the non-recursive one; a list of definitions; and the expression which is the body of the `let(rec)`. We choose to represent `isRec` as a boolean variable, and we define the two boolean values as follows:

```
> isRec        == bool
> recursive     = True
> nonRecursive = False
```

Each definition is just a pair of the variable name being bound and the expression to which it is bound. We define two useful functions which each take a list of definitions: `bindersOf` picks out the list of variables bound by the definitions, and `rhssOf` (short for 'right-hand sides of') extracts the list of right-hand sides to which they are bound.

```
> bindersOf       :: [(*,**)] -> [*]
> bindersOf defns =  [name | (name, rhs) <- defns]


> rhssOf        :: [(*,**)] -> [**]
> rhssOf defns  =  [rhs  | (name, rhs) <- defns]
```

`case` expressions have an expression to analyse, and a list of alternatives. Each alternative contains a tag, a list of the bound variables and the expression to the right of the arrow.

```
> alter * == (num, [*], expr *)
> coreAlt == alter name
```

We take the opportunity to define a useful function on expressions, a boolean-valued function, `isAtomicExpr`, which identifies 'atomic' expressions; that is, expressions with no internal structure:

```
> isAtomicExpr :: expr * -> bool
> isAtomicExpr (EVar v) = True
> isAtomicExpr (ENum n) = True
> isAtomicExpr e        = False
```

Finally, a Core-language program is just a list of supercombinator definitions:

```
> program * == [scDefn *]
> coreProgram == program name
```

A supercombinator definition contains the name of the supercombinator, its arguments and its body:

```
> scDefn * == (name, [*], expr *)
> coreScDefn == scDefn name
```

The argument list might be empty, in the case of a supercombinator with no arguments.

We conclude with a small example. Consider the following small program.

```
main = double 21 ;
double x = x+x
```

This program is represented by the following Miranda expression, of type `coreProgram`:

```
[("main",   [],    (EAp (EVar "double") (ENum 21))),
 ("double", ["x"], (EAp (EAp (EVar "+") (EVar "x")) (EVar "x")))
]
```

1.4 A small standard prelude

Miranda has a *standard prelude* which contains definitions of various useful functions (such as `map`, `foldr` and so on) which are always available. We will do the same for the Core language, by providing the following standard definitions:

```
I x = x ;
K  x y = x ;
K1 x y = y ;
S f g x = f x (g x) ;
compose f g x = f (g x) ;
twice f = compose f f
```

This ‘standard prelude’ is necessarily rather small, because we want it to work for *all* of our implementations, including the most primitive ones which will lack arithmetic and facilities for manipulating data structures. All that is available in the simplest implementations is function application!

The following definition for `preludeDefs`, which will be used throughout the book, embodies these definitions:

```
> preludeDefs :: coreProgram
> preludeDefs
>   = [ ("I", ["x"], EVar "x"),
>       ("K", ["x","y"], EVar "x"),
>       ("K1",["x","y"], EVar "y"),
>       ("S", ["f","g","x"], EAp (EAp (EVar "f") (EVar "x"))
>                                (EAp (EVar "g") (EVar "x"))),
>       ("compose", ["f","g","x"], EAp (EVar "f")
>                                       (EAp (EVar "g") (EVar "x"))),
>       ("twice", ["f"], EAp (EAp (EVar "compose") (EVar "f")) (EVar "f")) ]
```

1.5 A pretty-printer for the Core language

Once we have a value of type `coreProgram` it is often convenient to be able to display it. Miranda's built-in features are not much help here. For example, if one types `preludeDefs` in response to the Miranda prompt, the output produced is rather hard to understand. (Try it.)

What we require is a ‘pretty-printing’ function `pprint`, with type

```
> pprint :: coreProgram -> [char]
```

Then we could type `pprint preludeDefs`, and expect to get a list of characters which, when printed, looks like a nicely formatted version of `preludeDefs`. Our goal in this section is to write such a function.

When the result of a program is a list, Miranda usually prints out the list items separated by commas and surrounded by brackets. But in the special case when the result of the program is of type `[char]`, Miranda displays the list ‘all squashed up’, without square brackets and commas. For example, the value `"Hi\nthere"` is displayed as

```
Hi
there
```

and not as

```
['H', 'i', '\n', 't', 'h', 'e', 'r', 'e']
```

In this way, `pprint` can have complete control over the output format.

We will need some of the utility functions defined in Appendix A, so we import them using the `%include` directive:

```
> %include "utils"
```

1.5.1 Pretty-printing using strings

Let us first concentrate on Core-language expressions. It looks as though we require a pretty-printing function, `pprExpr`, defined something like this:

```
> pprExpr :: coreExpr -> [char]
> pprExpr (ENum n) = shownum n          || shownum is a Miranda built-in
> pprExpr (EVar v) = v
> pprExpr (EAp e1 e2) = pprExpr e1 ++ " " ++ pprAExpr e2
```

(We have deliberately left out many of the cases for `pprExpr` for the moment.) `pprAExpr` has the same type as `pprExpr`, but differs from it by placing parentheses around the expression unless it is a variable or number.

```
> pprAExpr :: coreExpr -> [char]
> pprAExpr e = pprExpr e,                        isAtomicExpr e
> pprAExpr e = "(" ++ pprExpr e ++ ")",          otherwise
```

One can proceed in this fashion, but there is a serious problem with doing so. The pretty-printer uses the list append function, `++`, a great deal. This can give very nasty performance, as the following example shows. Consider the expression

```
(xs1 ++ xs2) ++ xs3
```

The inner `++` takes time proportional to `#xs1`[2], but then the outer `++` takes time proportional to the length of `xs1++xs2`, so the total time taken is $(2 * \texttt{\#xs1}) + \texttt{\#xs2}$. In general, if we added more lists to this nested append, the cost can be quadratic in the length of the result! Of course, if we bracket the expression the other way, the cost is linear in the length of the result, but unfortunately we cannot guarantee this in a pretty-printer.

To demonstrate this effect, we will first write a function `mkMultiAp`, which makes it easy for us to build sample expressions of a given size. The call (`mkMultiAp` $\mathbf{n}\ \mathbf{e}_1\ \mathbf{e}_2$) generates a `coreExpr` representing the expression

$$\mathbf{e}_1\ \underbrace{\mathbf{e}_2\ \mathbf{e}_2\ \ldots\ \mathbf{e}_2}_{\mathbf{n}}$$

[2] The `#` function is a standard Miranda function for taking the length of a list.

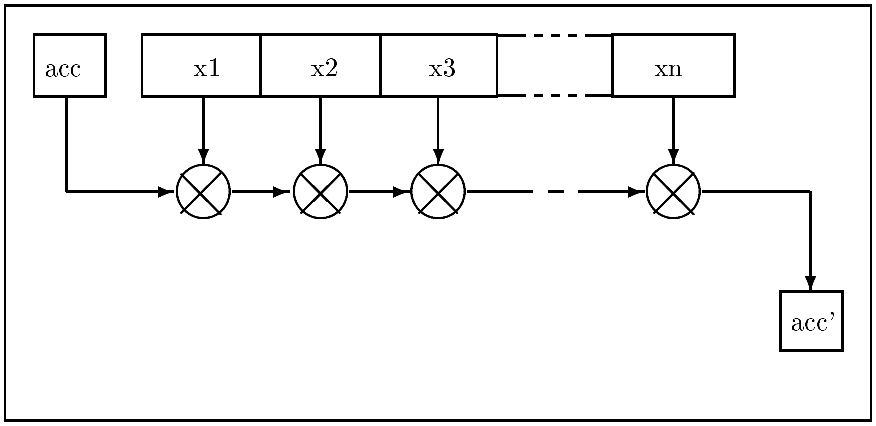


Figure 1.2: An illustration of `foldll` $\otimes$ **acc** $[\mathbf{x}_1, \ldots, \mathbf{x_n}]$

```
> mkMultiAp :: num -> coreExpr -> coreExpr -> coreExpr
> mkMultiAp n e1 e2 = foldll EAp e1 (take n e2s)
>                     where
>                     e2s = e2 : e2s
```

In this definition, `take` is a Miranda standard function which takes the first **n** elements of a list, discarding the rest of the list. The function `foldll` is a standard function, defined in Appendix A[3]. Given a dyadic function $\otimes$, a value **acc** and a list **xs** $= [\mathbf{x}_1, ..., \mathbf{x_n}]$, `foldll` $\otimes$ **acc xs** computes $\mathbf{acc}'$, where

$$\mathbf{acc}' = (\ldots((\mathbf{acc} \ \otimes \ \mathbf{x}_1) \ \otimes \ \mathbf{x}_2) \ \otimes \ \ldots \mathbf{x_n})$$

This is illustrated by Figure 1.2. In `mkMultiAp`, `foldll` is used to build a left-branching chain of `EAp` nodes. The initial accumulator **acc** is `e1`, and the combining function $\otimes$ is the `EAp` constructor. Finally, `e2s` is the infinite list `[e2, e2, ...]`; only its first `n` elements are used by `take`.

Exercise 1.1. Measure the number of Miranda steps required to compute

```
# (pprExpr (mkMultiAp n (EVar "f") (EVar "x")))
```

for various values of `n`. (You can use the Miranda directive `/count` to tell Miranda to print execution statistics. We take the length of the result so that the screen does not fill up with a huge printout.) Sketch a graph which shows how the execution cost rises with `n` and check that it is roughly quadratic in `n`.

1.5.2 An abstract data type for pretty-printing

A pretty-printer whose cost is quadratic in the size of the program to be printed is clearly unacceptable, so we had better find a way around it.

We can separate this problem into two parts: *'what operations do we want to perform?'*, and *'what is an efficient way to perform them?'*. In common with other languages, Miranda provides a way to make this distinction clear by introducing an *abstract data type*.

[3]We use `foldll` rather than the Miranda standard function `foldl` because different versions of Miranda have different definitions for `foldl`.

```
> abstype iseq with
>       iNil     :: iseq                    || The empty iseq
>       iStr     :: [char] -> iseq          || Turn a string into an iseq
>       iAppend  :: iseq -> iseq -> iseq    || Append two iseqs
>       iNewline :: iseq                    || New line with indentation
>       iIndent  :: iseq -> iseq            || Indent an iseq
>       iDisplay :: iseq -> [char]          || Turn an iseq into a string
```

The `abstype` keyword introduces an abstract data type, `iseq`. It is followed by the *interface* of the data type; that is, the operations which can be performed on the data type `iseq` and their type of each operation.

Given such a data type, we rewrite `pprExpr` to return an `iseq` instead of a list of characters:

```
> pprExpr :: coreExpr -> iseq
> pprExpr (ENum n) = iStr (shownum n)          || shownum is a Miranda built-in
> pprExpr (ENum n) = iStr (show n)
> pprExpr (EVar v) = iStr v
> pprExpr (EAp e1 e2) = (pprExpr e1) $iAppend (iStr " ") $iAppend (pprAExpr e2)
```

We have simply replaced `++` by `iAppend`[4], and added an `iStr` around literal strings.

What are the differences between an `iseq` and a list of characters? Firstly, we aim to produce an implementation of `iAppend` which does not have the unexpected quadratic behaviour of list append. Secondly, `iseq` provides new operations `iIndent` and `iNewline` which will be useful for controlling indentation. The idea is that `iIndent` indents its argument to line up with the current column; it should work even if its argument spreads over many lines, and itself contains calls to `iIndent`. `iNewline` stands for a newline followed by a number of spaces determined by the current level of indentation.

As an example of how `iIndent` and `iNewline` might be used, let us extend `pprExpr` to handle `let` and `letrec` expressions:

```
> pprExpr (ELet isrec defns expr)
>   = iConcat [  iStr keyword, iNewline,
>                iStr "  ",iIndent (pprDefns defns),iNewline,
>                iStr "in ",pprExpr expr ]
>     where
>     keyword = "let",           ~isrec
>             = "letrec",        isrec


> pprDefns :: [(name,coreExpr)] -> iseq
> pprDefns defns = iInterleave sep (map pprDefn defns)
>                  where
>                  sep = iConcat [ iStr ";", iNewline ]
```

[4]In Miranda, writing a dollar sign in front of an identifier turns it into an infix operator, allowing us to write `iAppend` between its arguments, instead of in front of them. Such infix operators are right-associative.

```
> pprDefn :: (name, coreExpr) -> iseq
> pprDefn (name, expr)
>   = iConcat [ iStr name, iStr " = ", iIndent (pprExpr expr) ]
```

To make the definitions more legible, we have used two new functions, `iConcat` and `iInterleave`, with the types

```
> iConcat     :: [iseq] -> iseq
> iInterleave :: iseq -> [iseq] -> iseq
```

`iConcat` takes a list of `iseq`s and uses `iAppend` to concatenate them into a single `iseq`. `iInterleave` is similar to `iConcat` except that it interleaves a specified `iseq` between each adjacent pair.

Exercise 1.2. Define `iConcat` and `iInterleave` in terms of `iAppend` and `iNil`.

In general, all our pretty-printing functions will return an `iseq`, and we apply `iDisplay` just once at the top level, to the `iseq` representing the entire thing we want to display:

```
> pprint prog = iDisplay (pprProgram prog)
```

Exercise 1.3. Add a further equation to `pprExpr` to handle `case` and lambda expressions, and write definitions for `pprAExpr` and `pprProgram` in the same style.

1.5.3 Implementing `iseq`

Now we come to the *implementation* of the `iseq` type. We begin by making an implementation that ignores all indentation. To implement the abstract data type we must say what type is used to represent an `iseq`:

```
> iseq == iseqRep
```

```
> iseqRep ::= INil
>           | IStr [char]
>           | IAppend iseqRep iseqRep
```

The first declaration says that the type `iseqRep` is used to represent an `iseq`, while the second declares `iseqRep` to be an algebraic data type with the three constructors `INil`, `IStr` and `IAppend`.

The general idea of this particular representation is to postpone all the work until the eventual call of `iDisplay`. The operations `iNil`, `iStr` and `iAppend` all just use the relevant constructor:

```
> iNil              = INil
> iAppend seq1 seq2 = IAppend seq1 seq2
> iStr str             = IStr str
```

Since we are ignoring indentation, `iIndent` and `iNewline` are defined trivially. We will improve them in the next section.

```
> iIndent seq = seq
> iNewline = IStr "\n"
```

All the interest lies in the operation `iDisplay` which turns an `iseq` into a list of characters. The goal is that it should only take time linear in the size of the `iseq`. It turns out to be convenient to define `iDisplay` in terms of a more general function, `flatten`:

```
> flatten :: [iseqRep] -> [char]
>
> iDisplay seq = flatten [seq]
```

The function `flatten` takes a *list* of `iseqRep`s, and returns the result of concatenating each of the `iseqRep`s in the list. The reason for having this list is that is allows us to accumulate a list of pending work, as we will soon see. Notice that `flatten` manipulates the *representation* type `iseqRep`, rather than the *abstract* type `iseq`.

We define `flatten` by case analysis on its argument, which we call the *work-list*. If the work-list is empty, we are done:

```
> flatten [] = ""
```

Otherwise, we work by doing case analysis on the first element of the work-list. The `INil` case just pops an item from the work-list:

```
> flatten (INil : seqs) = flatten seqs
```

The `IStr` case works by appending the specified string with the result of flattening the rest of the work-list:

```
> flatten (IStr s : seqs) = s ++ (flatten seqs)
```

So far, the fact that `flatten` takes a list has not helped us much. The justification for the list argument can be seen more clearly when we deal with `IAppend`; all that need be done is to push one more item onto the front of the work-list:

```
> flatten (IAppend seq1 seq2 : seqs)  = flatten (seq1 : seq2 : seqs)
```

Exercise 1.4. What is the cost of `flatten` in terms of the size of the `iseq`?

Change `pprExpr` to use `iseq` as indicated above, and measure the effect of the new implementation using the same experiment as in the previous exercise. Remember to apply `iDisplay` to the result of `pprExpr`.

Exercise 1.5. The key advantage of using an abstract data type is that one can change the *implementation* of the ADT without affecting its *interface*. As an example of this, redefine `iAppend` so that it returns a simplified result if either of its arguments is `INil`.

1.5.4 Layout and indentation

So far we have only given a rather trivial interpretation to the `iIndent` operation, and we now turn to improving it. In the same spirit as before, we first expand the `iseqRep` type with an extra two constructors, `IIndent` and `INewline`, and redefine their operations to use these constructors:

```
> iseqRep ::= INil
>           | IStr [char]
>           | IAppend iseqRep iseqRep
>           | IIndent iseqRep
>           | INewline
>
> iIndent seq = IIndent seq
> iNewline    = INewline
```

We must then make `flatten` more powerful. Firstly, it needs to keep track of the current column, and secondly, its work-list must consist of `(iseq, num)` pairs, where the number gives the indentation required for the corresponding `iseq`:

```
> flatten :: num                          || Current column; 0 for first column
>            -> [(iseqRep, num)]          || Work list
>            -> [char]                    || Result
```

We need to change `iDisplay` to initialise `flatten` appropriately:

```
> iDisplay seq = flatten 0 [(seq,0)]
```

The interesting case for `flatten` is when we deal with `INewline`, because this is where we need to perform indentation[5]:

```
> flatten col ((INewline, indent) : seqs)
> = "\n" ++ (spaces indent) ++ (flatten indent seqs)
```

Notice that the recursive call to flatten has a current-column argument of `indent` since we have now moved on to a new line and added `indent` spaces.

The `IIndent` case simply sets the current indentation from the current column:

```
> flatten col ((IIndent seq, indent) : seqs)
>  = flatten col ((seq, col) : seqs)
```

Exercise 1.6. Add equations for `flatten` for `IAppend`, `IStr` and `INil`.

Try `pprExpr` on an expression involving an `ELet`, and check that the layout works properly.

Exercise 1.7. The pretty-printer will go wrong if a newline character `'\n'` is embedded in a string given to `IStr`. Modify `iStr` to check for this, replacing the newline character by a use of `INewline`.

[5] `spaces` is a standard Miranda function which returns a list of a specified number of space characters.

1.5.5 Operator precedence

As discussed in Section 1.3, the `coreExpr` type has no construct for infix operator applications. Instead, such applications are expressed in prefix form, just like any other function application. It would be nice if our pretty-printer recognised such applications, and printed them in infix form. This is easily done by adding extra equations to `pprExpr` of the form

```
pprExpr (EAp (EAp (EVar "+") e1) e2)
= iConcat [   pprAExpr e1, iStr " + ", pprAExpr e2 ]
```

This still does not do a very good job, because it inserts too many parentheses. Would you prefer to see the expression

```
x + y > p * length xs
```

or the fully parenthesised version?

```
(x + y) > (p * (length xs))
```

The easiest way to achieve this is to give `pprExpr` an extra argument which indicates the precedence level of its context, and then use this to decide whether to add parentheses around the expression it produces. (The function `pprAExpr` now becomes redundant.)

Exercise 1.8. Make these changes to `pprExpr` and test them.

1.5.6 Other useful functions on `iseq`

Later on it will be useful to have a few more functions which work on `iseq`s. They are all defined in terms of the `iseq` interface functions, so the implementation can be changed without altering any of these definitions.

`iNum` maps a number to an `iseq` and `iFWNum` does the same except that the result is left-padded with spaces to a specified width:

```
> iNum :: num -> iseq
> iNum n = iStr (shownum n)      || shownum is a Miranda built-in


> iFWNum :: num -> num -> iseq
> iFWNum width n
>   = iStr (spaces (width - #digits) ++ digits)
>     where
>     digits = shownum n
```

(If the number is wider than the width required, a negative number will be passed to `spaces`, which then returns the empty list. So the net effect is to return a field just wide enough to contain the number.) `iLayn` lays out a list, numbering the items and putting a newline character after each, just as the standard function `layn` does.

```
> iLayn :: [iseq] -> iseq
> iLayn seqs = iConcat (map lay_item (zip2 [1..] seqs))
>              where
>              lay_item (n, seq)
>                = iConcat [ iFWNum 4 n, iStr ") ", iIndent seq, iNewline ]
```

1.5.7 Summary

Our pretty-printer still has its shortcomings. In particular, a good pretty-printer will lay things out on one line if they fit, and over many lines if they do not. It is quite possible to elaborate the `iseq` data type so that it can do this, but we will not do so here.

The `iseq` type is useful for pretty-printing data other than programs, and we will use it for a variety of purposes throughout the book.

There are two general points we would like to bring out from this section:

- It is very often helpful to separate the *interface* of an abstract data type from its *implementation*. Miranda provides direct support for this abstraction, by ensuring the functions over the abstract type do not inspect the representation.
- The definition of `iDisplay` in terms of `flatten` exemplifies a very common technique called *generalisation*. We often define the function we really want in terms of a simple call to a more general function. This is usually because the more general function carries around some extra arguments which it needs to keep the book-keeping straight.

  It is hard to make general statements about when generalisation is an appropriate technique; indeed, working out a good generalisation is often the main creative step in writing any program. However, there are plenty of examples of generalisation in this book, which we hope will help to convey the idea.

1.6 A parser for the Core language

We will want to run each of our implementations on a variety of Core programs. This means that we want a way of taking a file containing the Core program in its concrete syntax, and parsing it to a value of type `coreProgram`.

Writing parsers is generally rather tiresome, so much so that great effort has been devoted to building tools which accept a grammar and write a parser for you. The Unix Yacc utility is an example of such a parser generator. In a functional language, however, it is quite easy to write a simple parser, and we will do so in this section for the Core language. We split the task into three stages:

- First, we obtain the contents of the named file, as a list of characters. This is done by the built-in Miranda function `read`.
- Next, the *lexical analysis* function `lex` breaks the input into a sequence of small chunks, such as identifiers, numbers, symbols and so on. These small chunks are called *tokens*:

```
> lex :: [char] -> [token]
```

- Finally, the *syntax analysis* function `syntax` consumes this sequence of tokens and produces a `coreProgram`:

  ```
  > syntax :: [token] -> coreProgram
  ```

The full parser is just the composition of these three functions:

```
> parse :: [char] -> coreProgram
> parse = syntax . lex . read
```

The symbol '`.`' is Miranda's infix composition operator, which can be defined thus:

```
        (f . g) x  =  f (g x)
```

We could equivalently have defined `parse` without using composition, like this:

```
  parse filename = syntax (lex (read filename))
```

but it is nicer style to use composition, because it makes it particularly easy to see that we are defining `parse` as a pipeline of three functions.

1.6.1 Lexical analysis

We begin with the lexical analyser. We have not yet defined the type of a token. The easiest thing to begin with is to do no processing at all on the tokens, leaving them as (non-empty) strings:

```
> token == [char]               || A token is never empty
```

Now the lexical analysis itself. It should throw away white space (blanks, tabs, newlines):

```
> lex (c:cs) = lex cs, isWhiteSpace c
```

It should recognise numbers as a single token:

```
> lex (c:cs) = num_tok : lex rest_cs,          digit c
>              where
>              num_tok = c : takewhile digit cs
>              rest_cs = dropwhile digit cs
```

The standard function `digit` takes a character and returns `True` if and only if the character is a decimal digit. `takewhile` and `dropwhile` are both also standard functions; `takewhile` takes elements from the front of a list while a predicate is satisfied, and `dropwhile` removes elements from the front of a list while the predicate is satisfied. For example,

```
takewhile digit "123abc456"
```

is the list "123".

The lexical analyser should also recognise variables, which begin with an alphabetic letter, and continue with a sequence of letters, digits and underscores:

```
> lex (c:cs) = var_tok : lex rest_cs,          letter c
>              where
>              var_tok = c : takewhile isIdChar cs
>              rest_cs = dropwhile isIdChar cs
```

Here `letter` is a standard function like `digit` which returns `True` on alphabetic characters, and `isIdChar` is defined below.

If none of the above equations applies, the lexical analyser returns a token containing a single character.

```
> lex (c:cs) = [c] : lex cs
```

Lastly, when the input string is empty, `lex` returns an empty token list.

```
> lex [] = []
```

We conclude with the definitions of the auxiliary functions used above. (The operator '\/' is Miranda's boolean 'or' operation.)

```
> isIdChar c = letter c \/ digit c \/ (c = '_')
> isWhiteSpace c = member " \t\n" c
```

Exercise 1.9. Modify the lexical analyser so that it ignores comments as well as white space. Use the same convention that a comment is introduced by a double vertical bar, `||`, and extend to the end of the line.

Exercise 1.10. The lexical analyser does not currently recognise two-character operators, such as `<=` and `==`, as single tokens. We define such operators by giving a list of them:

```
> twoCharOps = ["==", "~=", ">=", "<=", "->"]
```

Modify `lex` so that it recognises members of `twoCharOps` as tokens. (The standard function `member` may be useful.)

Exercise 1.11. Since the lexical analysis throws away white space, the parser cannot report the line number of a syntax error. One way to solve this problem is to attach a line number to each token; that is, the type `token` becomes

```
token == (num, [char])
```

Alter the lexical analyser so that it does this. To do this you will need to add an extra parameter to `lex`, being the current line number.

1.6.2 Basic tools for parsing

In preparation for writing a parser for the Core language, we now develop some general-purpose functions to use when writing parsers. The techniques described below are well known [Fairbairn 1986, Wadler 1985], but make a rather nice demonstration of what can be done with functional programming. As a running example, we will use the following small grammar:

$$\begin{array}{lcl} \textbf{greeting} & \rightarrow & \textbf{hg}\ \textbf{person}\ \texttt{!} \\ \textbf{hg} & \rightarrow & \texttt{hello} \\ & \mid & \texttt{goodbye} \end{array}$$

where **person** is any token beginning with a letter.

Our general approach, which is very common in functional programming, is to try to build a big parser by glueing together smaller parsers. The key question is: what should the type of a parser be? It is a function which takes a list of tokens as its argument, and at first it appears that it should just return the parsed value. But this is insufficiently general, for two reasons.

1. Firstly, it must also return the remaining list of tokens. If, for example, we want to parse two items from the input, one after the other, we can apply the first parser to the input, but we must then apply the second parser to the remaining input returned by the first.

2. Secondly, the grammar may be ambiguous, so there is more than one way to parse the input; or the input may not conform to the grammar, in which case there is no way to successfully parse the input. An elegant way to accommodate these possibilities is to return a list of possible parses. This list is empty if there is no way to parse the input, contains one element if there is a unique way to parse it, and so on.

We can summarise our conclusion by defining the type of parsers using a type synonym, like this:

```
> parser * == [token] -> [(*, [token])]
```

That is, a parser for values of type `*` takes a list of tokens and returns a list of parses, each of which consists of a value of type `*` paired with the remaining list of tokens.

Now we are ready to define some small parsers. The function `pLit` ('lit' is short for 'literal') takes a string and delivers a parser which recognises only tokens containing that string, returning the string as the value of the parse:

```
> pLit :: [char] -> parser [char]
```

How does `pLit` work? It looks at the first token on the input and compares it with the desired string. If it matches, `pLit` returns a singleton list, indicating a single successful parse; if it does not match, `pLit` returns an empty list, indicating failure to parse[6]:

[6] This definition of `pLit` assumes that a token is just a string. If you have added line numbers to your tokens, as suggested in Exercise 1.11, then `pLit` will need to strip off the line number before making the comparison.

```
> pLit s (tok:toks) = [(s, toks)],     s = tok
>                   = [],              otherwise
> pLit s []         = []
```

The second equation takes care of the case where the input stream is empty. We can use `pLit` to define parsers which look for particular tokens in the input. For example, the expression

```
        pLit "hello" ["hello", "John", "!"]
```

evaluates to

```
        [("hello", ["John", "!"])]
```

Similarly, we define a parser `pVar` to parse a variable from the beginning of the input:

```
> pVar :: parser [char]
> pVar (tok:toks) = [(tok, toks)],     letter (hd tok)
>                 = [],                otherwise
> pVar (tok:toks) = isAlpha (hd tok) | [(tok, toks)]
>                 = otherWise | []
> pVar []         = []
```

`pVar` decides whether a token is a variable or not by looking at its first character. (The lexical analyser ensures that no token is empty.) Actually, this is not quite right, because it should not treat keywords as variables, but we will fix this problem later (Exercise 1.17).

The whole point of this development is to build bigger parsers by gluing together smaller ones, and we are now ready to do so. We will define a function `pAlt` ('alt' is short for 'alternative') which combines two parsers, say `p1` and `p2`. First it uses `p1` to parse the input, and then it uses `p2` to parse the *same* input; it returns all the successful parses returned by either `p1` or `p2`. So the type of `pAlt` is

```
> pAlt :: parser * -> parser * -> parser *
```

The actual definition of `pAlt` is delightfully simple. All it needs to is append the lists of parses returned by `p1` and `p2`:

```
> pAlt p1 p2 toks = (p1 toks) ++ (p2 toks)
```

For example, `pHelloOrGoodbye` is a parser which recognises either the token `"hello"` or `"goodbye"`:

```
> pHelloOrGoodbye :: parser [char]
> pHelloOrGoodbye = (pLit "hello") $pAlt (pLit "goodbye")
```

It is easy to see that `pAlt` corresponds directly to the vertical bar, |, of a BNF grammar (see Figure 1.1, for example). We need one other fundamental parser-combining function, `pThen`, which corresponds to the *sequencing* of symbols in a BNF grammar.

Like `pAlt`, `pThen` combines two parsers, say `p1` and `p2`, returning a bigger parser which behaves as follows. First, it uses `p1` to parse a value from the input, and then it uses `p2` to parse a second value from the remaining input. What value should `pThen` return from a successful parse? Presumably some combination of the values returned by `p1` and `p2`, so the right thing to do is to give `pThen` a third argument which is the value-combining function. So the type of `pThen` is:

```
> pThen :: (* -> ** -> ***) -> parser * -> parser ** -> parser ***
```

The definition of `pThen` makes use of a *list comprehension*:

```
> pThen combine p1 p2 toks
>   = [ (combine v1 v2, toks2) | (v1,toks1) <- p1 toks;
>                                (v2,toks2) <- p2 toks1]
```

The right-hand side of this equation should be read as follows:

> 'the list of pairs `(combine v1 v2, toks2)`,
> where `(v1,toks1)` is drawn from the list `p1 toks`,
> and `(v2,toks2)` is drawn from the list `p2 toks1`'.

With the aid of `pThen` we can make a parser for greetings:

```
> pGreeting :: parser ([char], [char])
> pGreeting = pThen mk_pair pHelloOrGoodbye pVar
>             where
>             mk_pair hg name = (hg, name)
```

For example, the expression

```
        pGreeting ["goodbye", "James", "!"]
```

would evaluate to

```
        [(("goodbye", "James"), ["!"])]
```

Notice that when writing `pGreeting` we did not need to think about the fact that `pHelloOrGoodbye` was itself a composite parser. We simply built `pGreeting` out of its component parsers, each of which has the same standard interface. We could subsequently change `pHelloOrGoodbye` without having to change `pGreeting` as well.

1.6.3 Sharpening the tools

We have now completed the basic tools for developing parsers. In this section we will develop them in a number of ways.

The definition of `pGreeting` given above is not quite right, because the grammar demands an exclamation mark after the person's name. We could fix the problem like this:

```
pGreeting = pThen keep_first
                  (pThen mk_pair pHelloOrGoodbye pVar)
                  (pLit "!")
            where
            keep_first hg_name exclamation = hg_name
            mk_pair hg name = (hg, name)
```

Since the final exclamation mark is always present, we have chosen not to return it as part of the parsed value; it is discarded by `keep_first`. This definition is rather clumsy, however. It would be more convenient to define a new function `pThen3`, so that we could write:

```
pGreeting = pThen3 mk_greeting
                  pHelloOrGoodbye
                  pVar
                  (pLit "!")
            where
            mk_greeting hg name exclamation = (hg, name)
```

Exercise 1.12. Give the type of `pThen3`, write down its definition, and test the new version of `pGreeting`. Similarly, write `pThen4`, which we will need later.

Another very common feature of grammars is to require zero or more repetitions of a symbol. To reflect this we would like a function, `pZeroOrMore`, which takes a parser, `p`, and returns a new parser which recognises zero or more occurrences of whatever `p` recognises. The value returned by a successful parse can be the list of the values returned by the successive uses of `p`. So the type of `pZeroOrMore` is

```
> pZeroOrMore :: parser * -> parser [*]
```

For example, a parser to recognise zero or more greetings is

```
> pGreetings = pZeroOrMore pGreeting
```

We can define `pZeroOrMore` by observing that it must either see one or more occurrences, or zero occurrences:

```
> pZeroOrMore p = (pOneOrMore p) $pAlt (pEmpty [])
```

Here, `pEmpty` is a parser which always succeeds, removing nothing from the input, returning the value it is given as its first argument:

```
> pEmpty :: * -> parser *
```

The function `pOneOrMore` has the same type as `pZeroOrMore`.

```
> pOneOrMore :: parser * -> parser [*]
```

Exercise 1.13. Write definitions for `pOneOrMore` and `pEmpty`. (Hint: you will find it convenient to call `pZeroOrMore` from `pOneOrMore`.) Test your definitions by using them to define a parser to recognise one or more greetings.

It is often convenient to process the values returned by successful parses. For example, suppose we wanted `pGreetings` to return the *number* of greetings rather than their content. To do this we would like to apply the length function, `#`, to the value returned by `pZeroOrMore`:

```
> pGreetingsN = (pZeroOrMore pGreeting) $pApply (#)
```

Here `pApply` is a new parser-manipulation function, which takes a parser and a function, and applies the function to the values returned by the parser:

```
> pApply :: parser * -> (* -> **) -> parser **
```

Exercise 1.14. Write a definition for `pApply`, and test it. (Hint: use a list comprehension.)

Another very common pattern in grammars is to look for one or more occurrences of a symbol, separated by some other symbol. For example, a **program** in Figure 1.1 is a sequence of one or more supercombinator definitions, separated by semicolons. We need yet another parser-building function, `pOneOrMoreWithSep`, whose type is

```
> pOneOrMoreWithSep :: parser * -> parser ** -> parser [*]
```

The second argument is the parser which recognises the separators, which are not returned as part of the result; that is why there is only one occurrence of `**` in the type.

Exercise 1.15. Define and test `pOneOrMoreWithSep`. It may help to think of the following grammar for **program**:

$$\begin{array}{lcl} \textbf{program} & \rightarrow & \textbf{sc programRest} \\ \textbf{programRest} & \rightarrow & \textbf{; program} \\ & | & \epsilon \end{array}$$

where ϵ is the empty string (corresponding to the `pEmpty` parser).

The parsers `pLit` and `pVar` are quite similar to each other: they both test for some property of the first token, and either fail (if it does not have the property) or succeed, returning the string inside the token (if it does have the property). We could generalise this idea by writing a parser `pSat` (where 'sat' is short for 'satisfies'), with the type

```
> pSat :: ([char] -> bool) -> parser [char]
```

`pSat` takes a function which tells whether or not the string inside the token has the desired property, and returns a parser which recognises a token with the property. Now we can write `pLit` in terms of `pSat`[7]:

[7] The expression `(= s)` is called a *section*. It is the partial application of the equality operator `=` to one argument `s`, producing a function which tests whether its argument is equal to `s`.

```
> pLit s = pSat (= s)
```

Exercise 1.16. Define `pSat` and test it. Write `pVar` in terms of `pSat` in a similar way to `pLit`.

`pSat` adds a useful level of modularity. For example, `pVar` currently recognises all alphabetic tokens as variables, but ultimately we might want it not to recognise language keywords (such as `let` and `case`) as variables.

Exercise 1.17. Modify the function passed to `pSat` in the definition of `pVar` above so that it does not treat strings in the list `keywords` as variables.

```
> keywords = ["let", "letrec", "case", "in", "of", "Pack"]
```

Exercise 1.18. As another example, use `pSat` to define a parser for numbers, with the type

```
> pNum :: parser num
```

`pNum` should use `pSat` to identify numeric tokens, and then `pApply` to convert the string to a number. (Miranda provides a standard function `numval` with type `[char] -> num` which can be used to do the hard work.)

There is an interesting performance problem associated with `pOneOrMore` and its related functions. Consider the following Core program:

```
f x = let x1 = x; x2 = x; ...; xn = x
      of x1
```

The idea is that we have a big `let` expression with definitions for `x1`, `x2`, ..., `xn` (the definitions are rather trivial, but they serve the purpose). This program has a syntax error: we have written '`of`' instead of '`in`' after the `let` expression.

Exercise 1.19. †Count how many Miranda steps it takes before the syntax error is reported, for `n` = $5, 10, 15, 20$ and so on. (Use Miranda's `/count` directive to get a display of execution statistics.) How fast does the parsing cost rise, in terms of `n`?

To get an idea why this happens, try evaluating:

```
pOneOrMore (pLit "x") ["x", "x", "x", "x", "x", "x"]
```

You should get a list of six possible parses. Based on this, can you work out why the parsing cost in the previous example rises so fast?

How can this problem be solved? (Hint: apart from the first one, are any of the parses returned by `pOneOrMore` useful? How could the extra ones be eliminated?)

1.6.4 Parsing the Core language

We are finally ready to define a parser for the Core language. First we deal with the 'wrapper' function, `syntax`. Recall that it takes a list of tokens and delivers a result of type `coreProgram`. It can do this by calling the parser `pProgram` which parses the non-terminal **program** (Figure 1.1), and then selecting the first complete parse it returns. If it returns no complete parse — that is, one in which the sequence of tokens remaining after the parse is empty — `syntax` produces a (horribly uninformative) error message.

```
> syntax = take_first_parse . pProgram
>          where
>          take_first_parse ((prog,[]) : others) = prog
>          take_first_parse (parse     : others) = take_first_parse others
>          take_first_parse other                = error "Syntax error"
```

The beauty of our parsing tools is that *we can write parsers by merely transliterating the grammar into Miranda.* For example, consider the productions for **program** and **sc** in Figure 1.1:

$$\begin{array}{lcll} \textbf{program} & \rightarrow & \textbf{sc}_1 \texttt{;} \ \ldots \texttt{;} \ \textbf{sc}_\mathbf{n} & (\mathbf{n} \geq 1) \\ \textbf{sc} & \rightarrow & \textbf{var} \ \textbf{var}_1 \ldots \textbf{var}_\mathbf{n} \ \texttt{=} \ \textbf{expr} & (\mathbf{n} \geq 0) \end{array}$$

We can transliterate these directly into Miranda:

```
> pProgram :: parser coreProgram
> pProgram = pOneOrMoreWithSep pSc (pLit ";")


> pSc :: parser coreScDefn
> pSc = pThen4 mk_sc pVar (pZeroOrMore pVar) (pLit "=") pExpr
```

Exercise 1.20. Write the function `mk_sc`. It takes four arguments returned by the four parsers used in `pSc`, and builds a value of type:

```
(name, [name], coreExpr)
```

It is a straightforward matter to complete the definitions for the rest of the grammar, apart from the productions for application and infix operators.

Exercise 1.21. Leaving these two productions out, complete the parser. A little care is needed for the parser `pAexpr`, which should have type `parser coreExpr`. The `pApply` function is required to wrap an `EVar` constructor around the value returned by `pVar`, and an `ENum` constructor around that returned by `pNum`.

Test your parser on the following program

```
f = 3 ;
g x y = let z = x in z ;
h x = case (let y = x in y) of
        <1> -> 2 ;
        <2> -> 5
```

You will find that the output becomes illegible as you run the parser on larger programs. To solve this, use the pretty-printing function `pprint` to format the output of your parser.

Exercise 1.22. Consider the program

```
f x y = case x of
        <1> -> case y of
                <1> -> 1;
        <2> -> 2
```

Does the alternative starting with `<2>` attach to the inner `case` or the outer one? Work out your answer, and see if your parser behaves as you expect. This is known as the 'dangling else' question.

Now we turn our attention to the two problems mentioned above.

1.6.5 Left recursion

The problem with applications is relatively easy to solve. The production for applications looks like this:

$$\textbf{expr} \;\to\; \textbf{expr aexpr}$$

If we simply transliterate this to

```
pExpr = pThen EAp pExpr pAexpr
```

then unfortunately `pExpr` will never terminate, because it keeps calling itself indefinitely. The problem is that **expr** appears as the first symbol in a production of **expr**; this is called *left recursion*. Our parsing tools simply cannot cope with left-recursive grammars. Fortunately, it is usually possible to transform the grammar so that it is no longer left-recursive, though the resulting grammar does not then reflect the structure of the result we are trying to construct. In this case, for example, we can simply use repetition, transforming the offending production to

$$\textbf{expr} \;\to\; \textbf{aexpr}_1 \ldots \textbf{aexpr}_\textbf{n} \qquad (\mathbf{n} \geq 1)$$

and now the parser `(pOneOrMore pAexpr)` can be used. The trouble is that this returns a list of expressions, rather than a single expression built with `EAp` constructors. We can solve this using `pApply`, giving the parser

```
    (pOneOrMore pAexpr) $pApply mk_ap_chain
```

Exercise 1.23. Define the appropriate function `mk_ap_chain` with type `[coreExpr] -> coreExpr`. Add the production for applications to your parser and test it.

1.6.6 Adding infix operators

The first problem with infix operators is that their precedence is implicit in the grammar of Figure 1.1. The standard way to make this explicit is to have several sorts of expression, as shown in Figure 1.3.

Notice the way that this grammar expresses the fact that `|` and `&` are right-associative, whereas relational operators are non-associative. Having to write out so many rules is rather tiresome, but we are only making explicit what we meant all along. But now the second problem arises: a parser implemented directly from these rules would be horribly inefficient! Consider the productions for **expr**1. A naive parser would attempt to recognise an **expr**2, and then look for a vertical bar `|`. If it did not find one (as will often be the case), *it will laboriously reparse the original input to look for an* **expr**2 *again.* Worse, each attempt to parse an **expr**2 may involve two attempts to parse an **expr**3, and hence four attempts to parse an **expr**4, and so on.

We want to share the parsing of the **expr**2 between the two productions, and this is not hard to do, by splitting the **expr**1 production into two:

$$\begin{array}{lcl} \textbf{expr1} & \to & \textbf{expr2 expr1c} \\ \textbf{expr1c} & \to & \texttt{|}\ \textbf{expr1} \\ & | & \epsilon \end{array}$$

expr	$\rightarrow$	`let` **defns** `in` **expr**	
	$\vert$	`letrec` **defns** `in` **expr**	
	$\vert$	`case` **expr** `of` **alts**	
	$\vert$	`\` $\mathbf{var_1 \ldots var_n}$ `.` **expr**	
	$\vert$	**expr1**	
expr1	$\rightarrow$	**expr2** `|` **expr1**	
	$\vert$	**expr2**	
expr2	$\rightarrow$	**expr3** `&` **expr2**	
	$\vert$	**expr3**	
expr3	$\rightarrow$	**expr4** **relop** **expr4**	
	$\vert$	**expr4**	
expr4	$\rightarrow$	**expr5** `+` **expr4**	
	$\vert$	**expr5** `-` **expr5**	
	$\vert$	**expr5**	
expr5	$\rightarrow$	**expr6** `*` **expr5**	
	$\vert$	**expr6** `/` **expr6**	
	$\vert$	**expr6**	
expr6	$\rightarrow$	$\mathbf{aexpr_1 \ldots aexpr_n}$	$(\mathbf{n} \geq 1)$

Figure 1.3: Grammar expressing operator precedence and associativity

Here ϵ stands for the empty string; the productions for **expr1c** say that an **expr1c** is either a vertical bar, `|`, followed by an **expr1**, or it is empty. We are almost there! The last question is: what is the type of a parser for **expr1c**. It cannot be of type `parser coreExpr`, because the phrase `|` **expr1** is only part of an expression, and the empty string ϵ is not an expression either. As usual, transforming the grammar has destroyed the structure.

The solution is fairly easy. We define a new data type `partialExpr`, like this

```
> partialExpr ::= NoOp | FoundOp name coreExpr
```

Now we can define the parser for **expr1c** like this:

```
> pExpr1c :: parser partialExpr
> pExpr1c = (pThen FoundOp (pLit "|") pExpr1) $pAlt (pEmpty NoOp)
```

The parser for **expr1** takes apart the intermediate result returned by `pExpr1c`:

```
> pExpr1 :: parser coreExpr
> pExpr1 = pThen assembleOp pExpr2 pExpr1c


> assembleOp e1 NoOp = e1
> assembleOp e1 (FoundOp op e2) = EAp (EAp (EVar op) e1) e2
```

Exercise 1.24. Transform the grammar along the lines suggested, transliterate the changes into Miranda code, and test the resulting parser.

1.6.7 Summary

The grammars that can be handled efficiently by our library of parser-building functions are called LL(1) grammars, exactly the same class that can be dealt with by conventional recursive-descent parsers [Aho *et al.* 1986].

Using the library we can easily write very concise parsers. This is an important and useful property, because almost any program has an input language of some sort, which has to be parsed by the program.

There are various things we have to take care about (left recursion, operator precedence, sharing), but exactly the same issues arise in any recursive-descent parser, regardless of the language in which it is implemented.

Chapter 2

Template instantiation

This chapter introduces the simplest possible implementation of a functional language: a graph reducer based on *template instantiation*.

The complete source code for an initial version (Mark 1) is given, followed by a series of improvements and variations on the basic design. We begin with a review of graph reduction and template instantiation.

2.1 A review of template instantiation

We begin with a brief overview of template instantiation. This material is covered in more detail in Chapters 11 and 12 of [Peyton Jones 1987].

We recall the following key facts:

- A functional program is 'executed' by *evaluating an expression*.
- The expression is represented by a *graph*.
- Evaluation takes place by carrying out a sequence of *reductions*.
- A reduction replaces (or *updates*) a *reducible expression* in the graph by its reduced form. The term 'reducible expression' is often abbreviated to 'redex'.
- Evaluation is complete when there are no more redexes; we say that the expression is in *normal form*.
- At any time there may be more than one redex in the expression being evaluated, so there is a choice about which one to reduce next. Fortunately, whatever reduction sequence we choose, we will always get the same answer (that is, normal form). There is one caveat: some reduction sequences may fail to terminate.
- However, if any choice of redexes makes evaluation terminate, then the policy of always selecting the outermost redex will also do so. This choice of reduction order is called *normal order reduction*, and it is the one we will always use.

Thus the process of evaluation can be described as follows:

```
until there are no more redexes
        select the outermost redex
        reduce it
        update the (root of the) redex with the result
end
```

2.1.1 An example

As an example, consider the following Core-language program:

```
square x = x * x ;
main = square (square 3)
```

The program consists of a set of definitions, called *supercombinators*; `square` and `main` are both supercombinators. By convention, the expression to be evaluated is the supercombinator `main`. Hence, to begin with the expression to be evaluated is represented by the following rather trivial tree (remember that a tree is just a special sort of graph):

```
main
```

Now, since `main` has no arguments, it itself is a redex, so we replace it by its body:

```
main           reduces to      @
                              / \
                        square   @
                                / \
                          square   3
```

Applications are represented by `@` signs in these pictures and all subsequent ones.

Now the outermost redex is the outer application of `square`. To reduce a function application we replace the redex with an instance of the body of the function, substituting a pointer to the argument for each occurrence of the formal parameter, thus:

```
        @!      reduces to      @!
       / \                     / \
 square   @                   @   \
         / \                 / \___@
   square   3               *     / \
                           square   3
```

The root of the redex, which is overwritten with the result, is marked with a `!`. Notice that the inner `square 3` redex has become shared, so that the tree has become a graph.

In the definition of `square` the expression `x*x` (in which the `*` is written infix) is just short for `((* x) x)`, the application of `*` to two arguments. We use *currying* to write functions of several arguments in terms of one-argument applications: `*` is a function which, when applied to an argument `p`, gives a function which, when applied to another argument `q`, returns the product of `p` and `q`.

Now the only redex is the inner application of `square` to `3`. The application of `*` is not reducible because `*` requires its arguments to be evaluated. The inner application is reduced like this:

```
        @       reduces to      @
       / \                     / \
      @   \                   @   \
     / \___@!                / \___@
    *     / \               *     / \
    square   3                   @   \
                                / \___3
                               *
```

There is still only one redex, the inner multiplication. We replace the redex with the result of the multiplication, `9`:

```
        @       reduces to      @
       / \                     / \
      @   \                   @   \
     / \___@!                / \___9
    *     / \               *
         @   \
        / \___3
       *
```

Notice that by physically updating the root of the redex with the result, both arguments of the outer multiplication 'see' the result of the inner multiplication. The final reduction is simple:

```
        @       reduces to      81
       / \
      @   \
     / \___9
    *
```

2.1.2 The three steps

As we saw earlier, graph reduction consists of repeating the following three steps until a normal form is reached:

1. Find the next redex.

2. Reduce it.

3. Update the (root of the) redex with the result.

As can be seen from the example in the previous section, there are two sorts of redex, which are reduced in different ways:

Supercombinators. If the outermost function application is a supercombinator application, then it is certainly also a redex, and it can be reduced as described below (Section 2.1.4).

Built-in primitives. If the outermost function application is the application of a built-in primitive, then the application may or may not be a redex, depending on whether the arguments are evaluated. If not, then the arguments must be evaluated. This is done using exactly the same process: repeatedly find the outermost redex of the argument and reduce it. Once this is done, we can return to the reduction of the outer application.

2.1.3 Unwinding the spine to find the next redex

The first step of the reduction cycle is to find the site of the next reduction to be performed; that is, the outermost reducible function application. It is easy to find the outermost function application (though it may not be reducible) as follows:

1. Follow the left branch of the application nodes, starting at the root, until you get to a supercombinator or built-in primitive. This left-branching chain of application nodes is called the *spine* of the expression, and this process is called *unwinding* the spine. Typically a *stack* is used to remember the addresses of the nodes encountered on the way down.
2. Now, check how many arguments the supercombinator or primitive takes and go back up that number of application nodes; you have now found the root of the outermost function application.

For example, in the expression `(f E1 E2 E3)`, where `f` takes two arguments, say, the outermost function application is `(f E1 E2)`. The expression and stack would look like this:

```
   Stack
-----------
  |  ---|-------> @
  -------        / \
  |  ---|-----> @!  E3
  -------      / \
  |  ---|---> @   E2
  -------    / \
  |  ---|-> f   E1
  -------
```

The (root of the) outermost function application is marked with a `!`.

If the result of an evaluation could be a partial application, as would be the case if `f` took four arguments instead of two, then step 2 above needs to be preceded by a check there are enough application nodes in the spine. If not, the expression has reached *weak head normal form* (WHNF). The sub-expressions `E1`, `E2` and `E3` might still contain redexes, but most evaluators will stop when they reach WHNF rather than trying to reduce the sub-expressions also. If the program has been type-checked, and the result is guaranteed to be a number, say, or a list, then this underflow check can be omitted.

Notice that we have only found the root of the outermost *function application*. It may or may not be a *redex* as well. If the function is a supercombinator, then it will certainly be a redex, but if it is a primitive, such as +, then it depends on whether its arguments are evaluated. If they are, we have found the outermost redex. If not, we have more work to do.

If a primitive requires the value of a currently unevaluated argument, we must evaluate the argument before the primitive reduction can proceed. To do this, we must put the current stack on one side, and begin with a new stack to reduce the argument, in the same way as before. This was the situation in the example of the previous section when we reached the stage

```
      @
     / \
    @   \
   / \___@
  *     / \
  square   3
```

We need to evaluate the argument `(square 3)` on a new stack. During this evaluation, we might again encounter a primitive with an unevaluated argument, so we would need to start a new evaluation again. We need to keep track of all the 'old' stacks, so that we come back to them in the right order. This is conveniently done by keeping a stack of stacks, called the *dump*. When we need to evaluate an argument, we push the current stack onto the dump; when we have finished an evaluation we pop the old stack off the dump.

Of course, in a real implementation we would not copy whole stacks! Since the 'new' stack will be finished with before the 'old' one is again required, the 'new' one could be built physically on top of the 'old' one. The dump stack would then just keep track of where the boundary between 'new' and 'old' was. Conceptually, though, the dump is a stack of stacks, and we will model it in this way.

2.1.4 Supercombinator redexes

A supercombinator redex is reduced by substituting the arguments into its body. More precisely:

> Supercombinator reduction. *A supercombinator redex is reduced by replacing the redex with an instance of the supercombinator body, substituting pointers to the actual arguments for corresponding occurrences of the formal parameters. Notice that the arguments are not copied; rather, by the device of using pointers to them, they are shared.*

A supercombinator body may contain `let` and `letrec` expressions. For example:

```
f x = let y = x*x
      in y+y
```

`let` and `letrec` expressions are treated as *textual descriptions of a graph.* Here, for example, is a possible use of the definition of `f`:

```
      @        reduces to      @
     / \                      / \
    f   3                    @   \
                            / \___@y
                           +     / \
                                @   \
                               / \___3
                              *
```

The `let` expression defines a sub-expression `x*x`, which is named `y`. The body of the `let` expression, `y+y`, uses pointers to the sub-expression in place of `y`. Thus ordinary expressions describe trees; `let` expressions describe acyclic graphs; and `letrec` expressions describe cyclic graphs.

2.1.5 Updates

After performing a reduction, we must update the root of the redex with the result, so that if the redex is shared (as it was in the example `(square (square 3))`) the reduction is only done once. This updating is the essence of *lazy evaluation*. A redex may not be evaluated at all but, if it is evaluated, the update ensures that the cost of doing so is incurred at most once.

Omitting the updates does not cause any errors; it will just mean that some expressions may be evaluated more than once, which is inefficient.

There is one case that requires a little care when performing updates. Consider the program

```
id x = x
f p = (id p) * p
main = f (sqrt 4)
```

After the `f` reduction has taken place, the graph looks like this:

```
        @
       / \
      @   \
     / \   \
    *   @   \
       / \___@
     id     / \
         sqrt   4
```

We assume `sqrt` is a built-in primitive for taking square roots. Now, suppose that the next redex selected is the first argument of the `*`, namely the application of `id`. (It might equally well be the second argument of `*`, since neither argument is in normal form, but we will suppose it is the first.) What should we overwrite the root of the redex with after performing the `id` reduction? *We should certainly not overwrite it with a copy of the* `(sqrt 4)` *application node, because then* `(sqrt 4)` *would be evaluated twice!*

The easiest way out of this dilemma is to add a new sort of graph node, an *indirection node*, which will be depicted as a `#` sign. An indirection node can be used to update the root of a redex to point to the result of the reduction:

```
        @       reduces to      @
       / \                     / \
      @   \                   @   \
     / \   \                 / \   \
    *   @   \               *   #   \
       / \___@                   \___@
     id     / \                     / \
        sqrt   4                sqrt   4
```

Section 12.4 of [Peyton Jones 1987] contains further discussion of the issues involved in updates.

2.1.6 Constant applicative forms†

Some supercombinators have no arguments; they are called *constant applicative forms*, or CAFs. For example, `fac20` is a CAF:

```
fac20 = factorial 20
```

The interesting thing about CAFs is that *the supercombinator itself is a redex.* We do not want to instantiate a new copy of `factorial 20` whenever `fac20` is called, because that would mean repeating the computation of `factorial 20`. Rather, the supercombinator `fac20` is the root of the `fac20`-reduction, and should be overwritten with the result of instantiating its body.

The practical consequence is that supercombinators should be represented by graph nodes, in order that they can be updated in the usual way. We will see this happening in practice in each of our implementations.

This concludes our review of graph reduction.

2.2 State transition systems

We now turn our attention to implementing graph reduction. We will describe each of our implementations using a *state transition system.* In this section we introduce state transition systems.

A state transition system is a notation for describing the behaviour of a sequential machine. At any time, the machine is in some *state*, beginning with a specified *initial state.* If the machine's state *matches* one of the *state transition rules*, the rule *fires* and specifies a new state for the machine. When no state transition rule matches, execution halts. If more than one rule matches, then one is chosen arbitrarily to fire; the machine is then *non-deterministic*. All our machines will be deterministic.

Here is a simple example of a state transition system used to specify a (rather inefficient) multiplication machine. The state is a quadruple $(\mathbf{n}, \mathbf{m}, \mathbf{d}, \mathbf{t})$. The numbers to be multiplied are $\mathbf{n}$ and $\mathbf{m}$, and the running total is $\mathbf{t}$, and the machine is initialised to the state $(\mathbf{n}, \mathbf{m}, 0, 0)$.

The operation of the machine is specified by two transition rules. The $\mathbf{d}$ component is repeatedly decremented towards zero while simultaneously incrementing $\mathbf{t}$, as specified by the first rule:

$$\begin{array}{|lllll|}\hline & \mathbf{n} & \mathbf{m} & \mathbf{d} & \mathbf{t} \\ \Longrightarrow & \mathbf{n} & \mathbf{m} & \mathbf{d}-1 & \mathbf{t}+1 \\ & \multicolumn{4}{l|}{\text{where } \mathbf{d} > 0} \\ \hline\end{array}$$

We always write transition rules with each component of the new state directly underneath the same component of the old state, so that it is easy to see which components have changed.

When $\mathbf{d}$ reaches zero it is initialised again to $\mathbf{n}$, and $\mathbf{m}$ is decremented, until $\mathbf{m}$ reaches zero. This is specified by the second rule:

$$\begin{array}{|lllll|}\hline & \mathbf{n} & \mathbf{m} & 0 & \mathbf{t} \\ \Longrightarrow & \mathbf{n} & \mathbf{m}-1 & \mathbf{n} & \mathbf{t} \\ & \multicolumn{4}{l|}{\text{where } \mathbf{m} > 0} \\ \hline\end{array}$$

The machine terminates when no rule applies. At this point it will be in a state $(\mathbf{n}, 0, 0, \mathbf{t})$, where $\mathbf{t}$ is the product of $\mathbf{n}$ and $\mathbf{m}$ from the initial state.

Exercise 2.1. Run the machine by hand starting with initial state $(2, 3, 0, 0)$, specifying which rule fires at each step. Verify that the final state is $(2, 0, 0, 6)$.

Exercise 2.2. An invariant of a sequence of states is a predicate which is true of all of the states. Find an invariant which expresses the relationship between the initial value of $\mathbf{n}$ and $\mathbf{m}$ (call them $\mathbf{N}$ and $\mathbf{M}$), and the current values of $\mathbf{m}$, $\mathbf{d}$ and $\mathbf{t}$. Hence *prove* the conjecture that the machine performs multiplication. To do the proof you need to show that

1. The invariant is true for the initial state.
2. If the invariant is true for a particular state, then it is true for its successor state.
3. Given the invariant and the termination condition ($\mathbf{m} = \mathbf{d} = 0$), then $\mathbf{t} = \mathbf{N} * \mathbf{M}$.
4. The machine terminates.

State transition systems are convenient for our purposes, because:

- They are sufficiently *abstract* that we do not get tangled up in very low-level details.
- They are sufficiently *concrete* that we can be sure we are not 'cheating' by hiding a lot of complexity in the rules.
- We can transliterate a state transition system directly into Miranda to give an executable implementation of the system.

To illustrate the last point, we will transliterate the multiplication machine into Miranda. We begin by giving a type synonym to define the type of a state in this machine:

```
> multState == (num, num, num, num)     || (n, m, d, t)
```

Next, the function `evalMult` takes a state and returns the list consisting of that state followed by all the states which follow it:

```
> evalMult :: multState -> [multState]
> evalMult state = [state],                              multFinal state
>                = state : evalMult (stepMult state),   otherwise
```

The function `stepMult` takes a non-final state and returns the next state. There is one equation for `stepMult` for each transition rule:

```
> stepMult (n, m, d, t) = (n, m,   d-1, t+1),   d>0
> stepMult (n, m, d, t) = (n, m-1, n,   t),     d=0
```

The function `multFinal` takes a state and tests whether the state is a final state:

```
> multFinal :: multState -> bool
```

Exercise 2.3. Define the function `multFinal`, and run the resulting machine on the initial state $(2,3,0,0)$, checking that the last state of the result list is $(2,0,0,6)$. You may find the standard function `layn` is useful to help lay out the results more legibly.

2.3 Mark 1: A minimal template instantiation graph reducer

We are now ready to begin the definition of a rather simple graph reducer. Even though it is simple, it contains many of the parts that more sophisticated graph reducers have, so it takes a few pages to explain.

2.3.1 Transition rules for graph reduction

The state of the template instantiation graph reduction machine is a quadruple

(stack, dump, heap, globals)

or *(s,d,h,f)* for short.

- The *stack* is a stack of *addresses*, each of which identifies a *node* in the heap. These nodes form the spine of the expression being evaluated. The notation $\mathbf{a}_1 : \mathbf{s}$ denotes a stack whose top element is $\mathbf{a}_1$ and the rest of which is $\mathbf{s}$.
- The *dump* records the state of the spine stack prior to the evaluation of an argument of a strict primitive. The dump will not be used at all in the Mark 1 machine, but it will be useful for subsequent versions.

- The *heap* is a collection of tagged *nodes*. The notation $\mathbf{h}[\mathbf{a} : \mathbf{node}]$ means that in the heap $\mathbf{h}$ the address $\mathbf{a}$ refers to the node $\mathbf{node}$.
- For each supercombinator (and later for each primitive), *globals* gives the address of heap node representing the supercombinator (or primitive).

A heap node can take one of three forms (for our most primitive machine):

- `NAp` $\mathbf{a}_1$ $\mathbf{a}_2$ represents the application of the node whose address is $\mathbf{a}_1$ to that whose address is $\mathbf{a}_2$.
- `NSupercomb` **args body** represents a supercombinator with arguments **args** and body **body**.
- `NNum` $\mathbf{n}$ represents the number $\mathbf{n}$.

There are only two state transition rules for this primitive template instantiation machine. The first one describes how to unwind a single application node onto the spine stack:

$$(2.1)\quad \boxed{\begin{array}{lrlll} & \mathbf{a} : \mathbf{s} & \mathbf{d} & \mathbf{h}[\mathbf{a} : \texttt{NAp}\ \mathbf{a}_1\ \mathbf{a}_2] & \mathbf{f} \\ \Longrightarrow & \mathbf{a}_1 : \mathbf{a} : \mathbf{s} & \mathbf{d} & \mathbf{h} & \mathbf{f} \end{array}}$$

(The heap component of the second line of this rule still includes the mapping of address $\mathbf{a}$ to `NAp` $\mathbf{a}_1$ $\mathbf{a}_2$, but we do not write it out again, to save clutter.) Repeated application of this rule will unwind the entire spine of the expression onto the stack, until the node on top of the stack is no longer a `NAp` node.

The second rule describes how to perform a supercombinator reduction.

$$(2.2)\quad \boxed{\begin{array}{lrlll} & \mathbf{a}_0 : \mathbf{a}_1 : \ldots : \mathbf{a_n} : \mathbf{s} & \mathbf{d} & \mathbf{h}[\mathbf{a}_0 : \texttt{NSupercomb}\ [\mathbf{x}_1, \ldots, \mathbf{x_n}]\ \mathbf{body}] & \mathbf{f} \\ \Longrightarrow & \mathbf{a_r} : \mathbf{s} & \mathbf{d} & \mathbf{h}' & \mathbf{f} \\ & \multicolumn{4}{l}{\text{where } (\mathbf{h}', \mathbf{a_r}) = \mathbf{instantiate\ body\ h\ f}[\mathbf{x}_1 \mapsto \mathbf{a}_1,\ \ldots,\ \mathbf{x_n} \mapsto \mathbf{a_n}]} \end{array}}$$

Most of the interest in this rule is hidden inside the function **instantiate**. Its arguments are:

- the expression to instantiate,
- a heap,
- the global mapping of names to heap addresses, $\mathbf{f}$, augmented by the mapping of argument names to their addresses obtained from the stack.

It returns a new heap and the address of the (root of the) newly constructed instance. Such a powerful operation is really at variance with the spirit of state transition systems, where each step is meant to be a simple atomic action, but that is the nature of the template instantiation machine. The implementations of later chapters will all have truly atomic actions!

Notice that the root of the redex is not itself affected by this rule; it is merely replaced on the stack by the root of the result. In other words, these rules describe a tree-reduction machine, which does *not* update the root of the redex, rather than a graph-reduction machine. We will improve on this later in Section 2.5.

2.3.2 Structure of the implementation

Now that we have a specification of our machine, we are ready to embark on its implementation.

Since we are writing the implementation in a functional language, we must write a function `run`, say, to do the job. What should its type be? It should take a filename, run the program therein, and print out the results, which might be either the final result or some kind of execution trace. So the type of `run` is given by the following type signature:

```
> run :: [char] -> [char]
```

Now we can think about how `run` might be built up. Running a program consists of four stages:

1. Parse the program from the expression found in a specified file. The `parse` function takes a filename and returns the parsed program.

   ```
   >     parse :: [char] -> coreProgram
   ```

2. Translate the program into a form suitable for execution. The `compile` function, which performs this task, takes a program and produces the initial state of the template instantiation machine:

   ```
   > compile :: coreProgram -> tiState
   ```

   `tiState` is the type of the state of the template instantiation machine. (The prefix '`ti`' is short for template instantiation.)

3. Execute the program, by performing repeated state transitions until a final state is reached. The result is a list of all the states passed through; from this we can subsequently either extract the final state, or get a trace of all the states. For the present we will restrict ourselves to programs which return a number as their result, so we call this execution function `eval`.

   ```
   > eval :: tiState -> [tiState]
   ```

4. Format the results for printing. This is done by the function `showResults`, which selects which information to print, and formats it into a list of characters.

   ```
   > showResults :: [tiState] -> [char]
   ```

The function `run` is just the composition of these four functions:

```
> run = showResults . eval . compile . parse
```

We will devote a subsection to each of these phases.

2.3.3 The parser

The source language, including the `parse` function, is defined in a separate module `language`, defined in Chapter 1. We make it available using the `%include` directive to import the module:

```
> %include "language"
```

2.3.4 The compiler

In this section we define the `compile` function. We will need the data types and functions defined in the `utils` module, so we use `%include` to make it available.

```
> %include "utils"
```

Now we need to consider the representation of the data types the compiler manipulates.

Data types

The compiler produces the initial state of the machine, which has type `tiState`, so the next thing to do is to define how machine states are represented, using a type synonym:

```
> tiState == (tiStack, tiDump, tiHeap, tiGlobals, tiStats)
```

The state of the machine is a quintuple whose first four components correspond exactly to those given in Section 2.3.1, and whose fifth component is used to accumulate statistics.

Next, we need to consider the representation of each of these components.

- The *spine stack* is just a stack of *heap addresses*:

  ```
  > tiStack == [addr]
  ```

  We choose to represent the stack as a list. The elements of the stack are members of the abstract data type `addr` defined in the `utils` module (Appendix A.1). They represent heap addresses, and by making them abstract we ensure that we can only use the operations provided on them by the `utils` module. Thus it is impossible for us to add one to an address, say, by mistake.

- The *dump* is not required until Section 2.6, but we make it part of the state already because adding it later would require many tiresome alterations to the state transition rules. For now we give it a trivial type definition, consisting of just a single constructor with no arguments.

  ```
  > tiDump ::= DummyTiDump
  > initialTiDump = DummyTiDump
  ```

- The *heap* is represented by the `heap` abstract data type defined in the `utils` module. We have to say what the heap contains, namely objects of type `node` (yet to be defined):

```
> tiHeap == heap node
```

Heap `node`s are represented by the following algebraic data type declaration, which corresponds to the list of possibilities given in Section 2.3.1:

```
> node ::= NAp addr addr                    || Application
>        | NSupercomb name [name] coreExpr  || Supercombinator
>        | NNum num                         || A number
```

The only difference is that we have added an extra field of type `name` to the `NSupercomb` constructor, which is used to hold the name of the supercombinator. This is used only for documentation and debugging purposes.

- The *globals* component associates each supercombinator name with the address of a heap node containing its definition:

```
> tiGlobals == assoc name addr
```

The `assoc` type is defined in the `utils` module, along with its operations (Appendix A.2). It is actually defined there as a type synonym (not an abstract data type) because it is so convenient to be able to manipulate associations using the built-in syntax for lists. There is a tension here between abstraction and ease of programming.

- The `tiStats` component of the state is not mentioned in the transition rules, but we will use it to collect run-time performance statistics on what the machine does. So that we can easily change what statistics are collected, we will make it an abstract type. To begin with, we will record only the number of steps taken:

```
> abstype tiStats
> with  tiStatInitial  :: tiStats
>       tiStatIncSteps :: tiStats -> tiStats
>       tiStatGetSteps :: tiStats -> num
```

The implementation is rather simple:

```
> tiStats == num
> tiStatInitial    = 0
> tiStatIncSteps s = s+1
> tiStatGetSteps s = s
```

A useful function `applyToStats` applies a given function to the statistics component of the state:

```
> applyToStats :: (tiStats -> tiStats) -> tiState -> tiState
> applyToStats stats_fun (stack, dump, heap, sc_defs, stats)
>  = (stack, dump, heap, sc_defs, stats_fun stats)
```

This completes our definition of the data types involved.

The compiler itself

The business of the compiler is to take a program, and from it create the initial state of the machine:

```
> compile program
>  = (initial_stack, initialTiDump, initial_heap, globals, tiStatInitial)
>    where
>    sc_defs = program ++ preludeDefs ++ extraPreludeDefs
>
>    (initial_heap, globals) = buildInitialHeap sc_defs
>
>    initial_stack = [address_of_main]
>    address_of_main = aLookup globals "main" (error "main is not defined")
```

Let us consider each of the definitions in the `where` clause in turn. The first, `sc_defs`, is just a list of all the supercombinator definitions involved in the program. Recall that `preludeDefs` was defined in Section 1.4 to be the list of standard supercombinator definitions which are always included in every program. `extraPreludeDefs` is a list of any further standard functions we may want to add; for the present it is empty:

```
> extraPreludeDefs = []
```

The second definition uses an auxiliary function, `buildInitialHeap`, to construct an initial heap containing an `NSupercomb` node for each supercombinator, together with an association list `globals` which maps each supercombinator name onto the address of its node.

Lastly, `initial_stack` is defined to contain just one item, the address of the node for the supercombinator `main`, obtained from `globals`.

Now we need to consider the definition of `buildInitialHeap`, which is a little tricky. We need to do something for each element of the list `sc_defs`, but what makes it awkward is that the 'something' involves heap allocation. Since each heap allocation produces a new heap, we need to find a way of passing the heap along from one element of `sc_defs` to the next. This process starts with the empty heap, `hInitial` (Appendix A.1).

We encapsulate this idea in a higher-order function `mapAccuml`, which we will use quite a lot in this book. `mapAccuml` takes three arguments: $\mathbf{f}$, the 'processing function'; $\mathbf{acc}$, the 'accumulator'; and a list $[\mathbf{x}_1, \ldots, \mathbf{x_n}]$. It takes each element of the input list, and applies $\mathbf{f}$ to it and the current accumulator. $\mathbf{f}$ returns a pair of results, an element of the result list and a new value for the accumulator. `mapAccuml` passes the accumulator along from one call of $\mathbf{f}$ to the next, and eventually returns a pair of results: $\mathbf{acc}'$, the final value of the accumulator; and the result list $[\mathbf{y}_1, \ldots, \mathbf{y_n}]$. Figure 2.1 illustrates this plumbing. The definition of `mapAccuml` is given in Appendix A.5.

In our case, the 'accumulator' is the heap, with initial value `hInitial`. The list $[\mathbf{x}_1, \ldots, \mathbf{x_n}]$ is the supercombinator definitions, `sc_defs`, while the result list $[\mathbf{y}_1, \ldots, \mathbf{y_n}]$ is the association of supercombinator names and addresses, `sc_addrs`. Here, then, is the definition of `buildInitialHeap`.

```
> buildInitialHeap :: [coreScDefn] -> (tiHeap, tiGlobals)
```

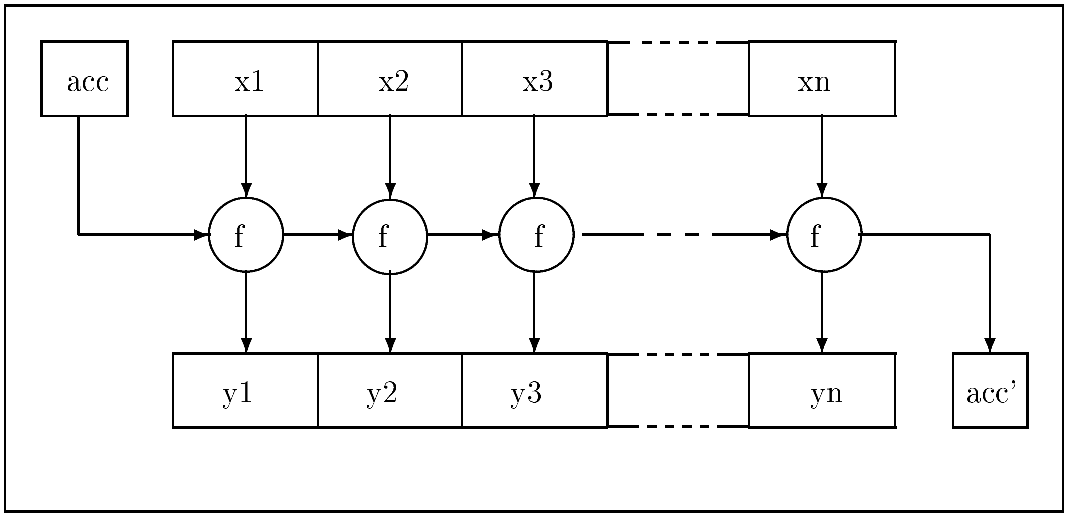


Figure 2.1: A picture of `mapAccuml` **f acc** $[\mathbf{x}_1, \ldots, \mathbf{x_n}]$

```
> buildInitialHeap sc_defs = mapAccuml allocateSc hInitial sc_defs
```

The 'processing function', which we will call `allocateSC`, allocates a single supercombinator, returning a new heap and a member of the `sc_addrs` association list.

```
> allocateSc :: tiHeap -> coreScDefn -> (tiHeap, (name, addr))
> allocateSc heap (name, args, body)
>  = (heap', (name, addr))
>    where
>    (heap', addr) = hAlloc heap (NSupercomb name args body)
```

That completes the definition of the compiler. Next, we turn our attention to the evaluator.

2.3.5 The evaluator

The evaluator `eval` takes an initial machine state, and runs the machine one step at a time, returning the list of all states it has been through.

`eval` always returns the current state as the first element of its result. If the current state is a final state, no further states are returned; otherwise, `eval` is applied recursively to the next state. The latter is obtained by taking a single step (using `step`), and then calling `doAdmin` to do any administrative work required between steps.

```
> eval state = state : rest_states
>              where
>              rest_states = [],                tiFinal state
>                          = eval next_state,   otherwise
>              next_state  = doAdmin (step state)
```

```
> doAdmin :: tiState -> tiState
> doAdmin state = applyToStats tiStatIncSteps state
```

Testing for a final state

The function `tiFinal` detects the final state. We are only finished if the stack contains a single object, and it is either a number or a data object.

```
> tiFinal :: tiState -> bool
>
> tiFinal ([sole_addr], dump, heap, globals, stats)
>  = isDataNode (hLookup heap sole_addr)
>
> tiFinal ([], dump, heap, globals, stats) = error "Empty stack!"
> tiFinal state = False              || Stack contains more than one item
```

Notice that the stack element is an address, which we need to look up in the heap before we can check whether it is a number or not. We should also produce a sensible error message if the stack should be empty (which should never happen).

Finally, we can define `isDataNode`:

```
> isDataNode :: node -> bool
> isDataNode (NNum n) = True
> isDataNode node     = False
```

Taking a step

The function `step` maps one state into its successor:

```
> step :: tiState -> tiState
```

It has to do case analysis on the node on top of the spine stack, so it extracts this node from the heap, and uses `dispatch` to call an appropriate function to do the hard work for each form of node.

```
> step state
>  = dispatch (hLookup heap (hd stack))
>    where
>    (stack, dump, heap, globals, stats) = state
>
>    dispatch (NNum n)                  = numStep state n
>    dispatch (NAp a1 a2)               = apStep  state a1 a2
>    dispatch (NSupercomb sc args body) = scStep  state sc args body
```

We can deal with the cases for numbers and applications with very little trouble. It is an error for there to be a number on top of the stack, since a number should never be applied as a function. (If it was the only object on the stack, execution will have been halted by `tiFinal`.)

```
> numStep :: tiState -> num -> tiState
> numStep state n = error "Number applied as a function!"
```

Dealing with an application node is described by the unwind rule (Rule 2.1), which can be translated directly into Miranda:

```
> apStep :: tiState -> addr -> addr -> tiState
> apStep (stack, dump, heap, globals, stats) a1 a2
>  = (a1 : stack, dump, heap, globals, stats)
```

Applying a supercombinator

To apply a supercombinator, we must instantiate its body, binding the argument names to the argument addresses found in the stack (Rule 2.2). Then we discard the arguments from the stack, including the root of the redex, and push the (root of the) result of the reduction onto the stack instead. (Remember, in this first version of the machine we are not performing updates.)

```
> scStep   :: tiState -> name -> [name] -> coreExpr -> tiState
> scStep (stack, dump, heap, globals, stats) sc_name arg_names body
>  = (new_stack, dump, new_heap, globals, stats)
>    where
>    new_stack = result_addr : (drop (#arg_names+1) stack)
>
>    (new_heap, result_addr) = instantiate body heap env
>    env = arg_bindings ++ globals
>    arg_bindings = zip2 arg_names (getArgs heap stack)
```

In order to apply supercombinators and primitives, we need an auxiliary function. The function `getArgs` takes a stack (which must consist of a supercombinator on top of a stack of application nodes), and returns a list formed from the argument of each of the application nodes on the stack.

```
> getArgs :: tiHeap -> tiStack -> [addr]
> getArgs heap (sc:stack)
>  = map get_arg stack
>    where get_arg addr = arg  where (NAp fun arg) = hLookup heap addr
```

The `instantiate` function takes an expression, a heap and an environment associating names with addresses. It creates an instance of the expression in the heap, and returns the new heap and address of the root of the instance. The environment is used by `instantiate` to specify the addresses to be substituted for supercombinators and local variables.

```
> instantiate :: coreExpr              || Body of supercombinator
>                -> tiHeap             || Heap before instantiation
>                -> assoc name addr    || Association of names to addresses
>                -> (tiHeap, addr)     || Heap after instantiation, and
>                                      || address of root of instance
```

The case for numbers is quite straightforward.

```
> instantiate (ENum n) heap env = hAlloc heap (NNum n)
```

The case for applications is also simple; just instantiate the two branches, and build the application node. Notice how we 'thread' the heap though the recursive calls to `instantiate`. That is, the first call to instantiate is given a heap and produces a new heap; the latter is given to the second call to instantiate, which produces yet another heap; the latter is the heap in which the new application node is allocated, producing a final heap which is returned to the caller.

```
> instantiate (EAp e1 e2) heap env
>  = hAlloc heap2 (NAp a1 a2) where (heap1, a1) = instantiate e1 heap  env
>                                   (heap2, a2) = instantiate e2 heap1 env
```

For variables, we simply look up the name in the environment we are given, producing a suitable error message if we do not find a binding for it.

```
> instantiate (EVar v) heap env
>  = (heap, aLookup env v (error ("Undefined name " ++ show v)))
```

`aLookup`, which is defined in Appendix A.2, looks up a variable in an association list, but returns its third argument if the lookup fails.

We postpone the question of instantiating constructors and `let(rec)` expressions by calling auxiliary functions `instantiateConstr` and `instantiateLet`, which each give errors for the present; later we will replace them with operational definitions. Lastly, the template machine is unable to handle `case` expressions at all, as we will see.

```
> instantiate (EConstr tag arity) heap env
>               = instantiateConstr tag arity heap env
> instantiate (ELet isrec defs body) heap env
>               = instantiateLet isrec defs body heap env
> instantiate (ECase e alts) heap env = error "Can't instantiate case exprs"


> instantiateConstr tag arity heap env
>             = error "Can't instantiate constructors yet"
> instantiateLet isrec defs body heap env
>             = error "Can't instantiate let(rec)s yet"
```

2.3.6 Formatting the results

The output from `eval` is a list of states, which are rather voluminous if printed in their entirety. Furthermore, since the heaps and stacks are abstract objects, Miranda will not print them at all. So the `showResults` function formats the output for us, using the `iseq` data type introduced in Section 1.5.

```
> showResults states
>  = iDisplay (iConcat [ iLayn (map showState states),
>                        showStats (last states)
>           ])
```

We display the state just by showing the contents of the stack. It is too tiresome to print the heap in its entirety after each step, so we will content ourselves with printing the contents of nodes referred to directly from the stack. The other components of the state do not change, so we will not print them either.

```
> showState :: tiState -> iseq
> showState (stack, dump, heap, globals, stats)
>  = iConcat [ showStack heap stack, iNewline ]
```

We display the stack, topmost element first, by displaying the address on the stack, and the contents of the node to which it points. Most of these nodes are application nodes, and for each of these we also display the contents of its argument node.

```
> showStack :: tiHeap -> tiStack -> iseq
> showStack heap stack
>  = iConcat [
>        iStr "Stk [",
>        iIndent (iInterleave iNewline (map show_stack_item stack)),
>        iStr " ]"
>    ]
>    where
>    show_stack_item addr
>     = iConcat [ showFWAddr addr, iStr ": ",
>                 showStkNode heap (hLookup heap addr)
>       ]


> showStkNode :: tiHeap -> node -> iseq
> showStkNode heap (NAp fun_addr arg_addr)
>  = iConcat [   iStr "NAp ", showFWAddr fun_addr,
>                iStr " ", showFWAddr arg_addr, iStr " (",
>                showNode (hLookup heap arg_addr), iStr ")"
>    ]
> showStkNode heap node = showNode node
```

`showNode` displays the value of a `node`. It prints only the name stored inside `NSupercomb` nodes, rather than printing the complete value; indeed this is the only reason the name is stored inside these nodes.

```
> showNode :: node -> iseq
> showNode (NAp a1 a2) = iConcat [ iStr "NAp ", showAddr a1,
>                                  iStr " ",    showAddr a2
>                        ]
> showNode (NSupercomb name args body) = iStr ("NSupercomb " ++ name)
> showNode (NNum n) = (iStr "NNum ") $iAppend (iNum n)
```

```
> showAddr :: addr -> iseq
> showAddr addr = iStr (show addr)
>
> showFWAddr :: addr -> iseq    || Show address in field of width 4
> showFWAddr addr = iStr (spaces (4-#str) ++ str)
>                   where
>                   str = show addr
```

`showStats` is responsible for printing out the accumulated statistics:

```
> showStats :: tiState -> iseq
> showStats (stack, dump, heap, globals, stats)
>  = iConcat [ iNewline, iNewline, iStr "Total number of steps = ",
>              iNum (tiStatGetSteps stats)
>    ]
```

Exercise 2.4. Test the implementation given so far. Here is a suitable test program:

```
main = S K K 3
```

The result should be the number 3. Invent a couple more test programs and check that they work. Remember, we have not yet defined any arithmetic operations!

Exercise 2.5. Modify `showState` so that it prints out the entire contents of the heap. (Hint: use `hAddresses` to discover the addresses of all the nodes in the heap.) In this way you can see how the heap evolves from one step to the next.

Exercise 2.6. `scStep` will fail if the supercombinator or primitive is applied to too few arguments. Add a suitable check and error message to `scStep` to detect this case.

Exercise 2.7. Modify your interpreter to collect more execution statistics. For example, you could accumulate:

- The number of reductions, perhaps split into supercombinator reductions and primitive reductions.
- The number of each kind of heap operation, especially allocations. The most convenient way to do this is to modify the `heap` abstract data type to accumulate this information itself, though this only works for heap operations which return a new heap as part of the result.

- The maximum stack depth.

Exercise 2.8. In the definition of `scStep`, the environment `env` which is passed to `instantiate` is defined as

```
env = arg_bindings ++ globals
```

What difference would it make if the arguments to `++` were reversed?

Exercise 2.9. (Slightly tricky.) You might think that the following definition for `eval` would be more obvious than the one given:

```
eval state = [state],                    tiFinal state
           = state : eval next_state,    otherwise
```

(where `next_state` is defined as before). Why is this an inferior definition? (Hint: think about what would happen if all the states were being formatted by `showResults`, and some error occurred when evaluating `tiFinal state`, such as an attempt to access a non-existent heap node. Would the state which caused the error be printed? If not, why not?)

2.4 Mark 2: `let(rec)` expressions

Our first enhancement is to make the machine capable of dealing with `let` and `letrec` expressions. As discussed in Section 2.1.4, the bodies of supercombinators may contain `let(rec)` expressions, which are regarded as textual descriptions of a graph.

It follows that the only change we have to make to our implementation is to enhance `instantiate`, so that it has an equation for the `ELet` constructor.

Exercise 2.10. Add an equation to `instantiate` for *non-recursive* `let` expressions. What you will need to do to instantiate `(ELet nonRecursive defs body)` is:

1. instantiate the right-hand side of each of the definitions in `defs`;
2. augment the environment to bind the names in `defs` to the addresses of the newly constructed instances;
3. call `instantiate` passing the augmented environment and the expression `body`.

This still only takes care of `let` expressions. The result of instantiating a `letrec` expression is a *cyclic* graph, whereas `let` expressions give rise to acyclic graphs.

Exercise 2.11. Copy your equation for the non-recursive `ELet` of `instantiate`, and modify it to work for the recursive case (or modify your definition to deal with both).

(Hint: do everything exactly as in the `let` case, except that in Step 1 pass the *augmented* environment (constructed in Step 2) to `instantiate`, instead of the *existing* environment.)

The hint in this exercise seems curious, because it requires the name-to-address bindings produced in Step 2 to be used as an input to Step 1. If you try this in Miranda it all works perfectly because, as in any non-strict functional language, the inputs to a function do not have to be evaluated before the function is called. In a real implementation we would have to do this trick 'by hand', by working out the addresses at which each of the (root) nodes in the `letrec` will be allocated, augmenting the environment to reflect this information, and then instantiating the right-hand sides.

Here is a test program, to see if your implementation works:

```
pair x y f = f x y ;
fst p = p K ;
snd p = p K1 ;
f x y = letrec
            a = pair x b ;
            b = pair y a
        in
        fst (snd (snd (snd a))) ;
main = f 3 4
```

The result should be 4. Can you figure out how this program works? (All will be revealed in Section 2.8.3.)

Exercise 2.12. Consider the program

```
main = letrec f = f x in f
```

What happens if you run this program? Could this problem ever arise in a strongly typed language such as Miranda?

2.5 Mark 3: Adding updating

So far our reduction machine does not perform any updates, so shared sub-expressions may be evaluated many times. As explained in Section 2.1.5 the easiest way to fix the problem is to update the root of the redex with an indirection node pointing to the result.

We can express this by modifying the state transition rule (2.2) for supercombinator redexes:

$$(2.3)\quad \boxed{\begin{array}{lrlll} & \mathbf{a_0 : a_1 : \ldots : a_n : s} & \mathbf{d} & \mathbf{h}[\mathbf{a_0} : \texttt{NSupercomb}\ [\mathbf{x_1}, \ldots, \mathbf{x_n}]\ \mathbf{body}] & \mathbf{f} \\ \Longrightarrow & \mathbf{a_r : s} & \mathbf{d} & \mathbf{h'}[\mathbf{a_n} : \texttt{NInd}\ \mathbf{a_r}] & \mathbf{f} \\ & \multicolumn{4}{l}{\text{where } (\mathbf{h'}, \mathbf{a_r}) = \mathbf{instantiate\ body\ h\ f}[\mathbf{x_1} \mapsto \mathbf{a_1},\ \ldots,\ \mathbf{x_n} \mapsto \mathbf{a_n}]} \end{array}}$$

The difference is that the heap $\mathbf{h'}$ returned by the **instantiate** function is further modified by overwriting the node $\mathbf{a_n}$ (the root of the redex) with an indirection to $\mathbf{a_r}$ (the root of the result, returned by **instantiate**). Notice that if the supercombinator is a CAF (see Section 2.1.6), then $\mathbf{n} = 0$ and the node to be modified is the supercombinator node itself.

One further modification is required. Since we may now encounter indirections during unwinding the spine, we need to add a new rule to cope with this case:

$$(2.4)\quad \boxed{\begin{array}{lrlll} & \mathbf{a : s} & \mathbf{d} & \mathbf{h}[\mathbf{a} : \texttt{NInd}\ \mathbf{a_1}] & \mathbf{f} \\ \Longrightarrow & \mathbf{a_1 : s} & \mathbf{d} & \mathbf{h} & \mathbf{f} \end{array}}$$

The address of the indirection node, **a**, is removed from the stack, just as if it had never been there.

There are several things we need to do to implement these new rules:

- Add a new node constructor, `NInd`, to the `node` data type. This gives the following revised definition:

```
> node ::= NAp addr addr                         || Application
>          | NSupercomb name [name] coreExpr    || Supercombinator
>          | NNum num                            || Number
>          | NInd addr                           || Indirection
```

  We need to add a new equation to `showNode` to take account of this extra constructor.

- Modify `scStep` to use `hUpdate` to update the root of the redex with an indirection to the result (Rule 2.3).

- Add an equation to the definition of `dispatch` to cope with indirections (Rule 2.4).

Exercise 2.13. Make the modifications to perform updates with indirection nodes. Try out the effect of your changes by running the following program on both the Mark 1 and Mark 3 versions of your reduction machine:

```
id x = x ;
main = twice twice id 3
```

(Recall that `twice` is defined in `preludeDefs` – Section 1.4.) Try to figure out what would happen by reducing it by hand first. What happens if you define `main` to be `twice twice twice id 3`?

2.5.1 Reducing the number of indirections

Often we will be updating the root of the redex with an indirection to a node newly created by `instantiate` (or, as we shall see, by a primitive). Under these circumstances, rather than use an indirection, it would be safe to build the root node of the result directly on top of the root of the redex. Because the root of the result is newly created, no sharing can be lost by doing this, and it saves building (and subsequently traversing) an extra indirection node.

We can do this by defining a new instantiation function, `instantiateAndUpdate`, which is just like `instantiate` except that it takes an extra argument, the address of the node to be updated with the result, and it does not return the address of the resulting graph.

```
> instantiateAndUpdate
>     :: coreExpr              || Body of supercombinator
>        -> addr               || Address of node to update
>        -> tiHeap             || Heap before instantiation
>        -> assoc name addr    || Association of parameters to addresses
>        -> tiHeap             || Heap after instantiation
```

Here, for example, is the definition of `instantiateAndUpdate` in the case when the expression is an application:

```
> instantiateAndUpdate (EAp e1 e2) upd_addr heap env
>  = hUpdate heap2 upd_addr (NAp a1 a2)
>    where
>    (heap1, a1) = instantiate e1 heap  env
>    (heap2, a2) = instantiate e2 heap1 env
```

Notice that the recursive instantiations are still performed by the old `instantiate`; only the root node needs to be updated.

Exercise 2.14. Complete the definition of `instantiateAndUpdate`. The following points need a little care:

- When the expression to be instantiated is a simple variable, you will still need to use an indirection. Why?
- Think carefully about the recursive instantiations in the equations for `let(rec)` expressions.

Modify `scStep` to call `instantiateAndUpdate` instead of `instantiate`, passing the root of the redex as the address of the node to be updated. Remove the update code from `scStep` itself.

Measure the effect of this modification on the number of reductions and the number of heap nodes allocated.

2.6 Mark 4: Adding arithmetic

In this section we will add arithmetic primitives. This will involve using the dump for the first time.

2.6.1 Transition rules for arithmetic

First of all we develop the state transition rules for arithmetic. We begin with negation, because it is a simple unary operation. The rules for other arithmetic operations are similar. Here is a plausible-looking rule when the argument is evaluated:

$$(2.5)\quad \boxed{\begin{array}{lrcll} & \mathbf{a}:\mathbf{a}_1:[\,] & \mathbf{d} & \mathbf{h}\left[\begin{array}{l}\mathbf{a}:\texttt{NPrim Neg}\\ \mathbf{a}_1:\texttt{NAp}\ \mathbf{a}\ \mathbf{b}\\ \mathbf{b}:\texttt{NNum}\ \mathbf{n}\end{array}\right] & \mathbf{f}\\ \Longrightarrow & \mathbf{a}_1:[\,] & \mathbf{d} & \mathbf{h}[\mathbf{a}_1:\texttt{NNum}\ (-\mathbf{n})] & \mathbf{f}\end{array}}$$

Notice that the rule specifies that the stack should contain *only* the argument to the negation operator, because anything else would be a type error.

Suppose that the argument is not evaluated: what should happen then? We need to evaluate the argument on a fresh stack (so that the evaluations do not get mixed up with each other) and, when this is complete, restore the old stack and try again. We need a way to keep track of the old stack, so we introduce the *dump* for the first time. The dump is just a stack of stacks.

The `Neg` rule to start an evaluation is like this:

$$(2.6)\quad \begin{array}{|lrcll|l|} \hline & \mathbf{a}:\mathbf{a}_1:[] & & \mathbf{d} & \mathbf{h}\left[\begin{array}{l} \mathbf{a}:\texttt{NPrim Neg} \\ \mathbf{a}_1:\texttt{NAp}\ \mathbf{a}\ \mathbf{b} \end{array}\right] & \mathbf{f} \\ \Longrightarrow & \mathbf{b}:[] & (\mathbf{a}:\mathbf{a}_1:[]):\mathbf{d} & \mathbf{h} & & \mathbf{f} \\ \hline \end{array}$$

This rule is used only if the previous one (which is a special case of this one, with the node at address **b** being an `NNum` node) does not apply.

Once the evaluation is complete, we need a rule to restore the old stack:

$$(2.7)\quad \begin{array}{|lcccl|} \hline & \mathbf{a}:[] & \mathbf{s}:\mathbf{d} & \mathbf{h}[\mathbf{a}:\texttt{NNum}\ \mathbf{n}] & \mathbf{f} \\ \Longrightarrow & \mathbf{s} & \mathbf{d} & \mathbf{h} & \mathbf{f} \\ \hline \end{array}$$

Once the old stack has been restored, the negation primitive will be found on top of the stack again, but this time the argument will be in normal form.

But we need to take care! The argument will indeed have been reduced to normal form, but the root node of the argument will have been updated, *so it may now be an indirection node.* Hence, the first rule for `Neg` will not see the `NNum` node directly. (For example, consider the expression `(negate (id 3))`.)

The easiest way around this is to add an extra transition rule just before the rule which unwinds an application node (Rule 2.1). In the special case where the argument of the application is an indirection, the rule updates the application with a new one whose argument points past the indirection:

$$(2.8)\quad \begin{array}{|lccl|l|} \hline & \mathbf{a}:\mathbf{s} & \mathbf{d} & \mathbf{h}\left[\begin{array}{l} \mathbf{a}:\texttt{NAp}\ \mathbf{a}_1\ \mathbf{a}_2 \\ \mathbf{a}_2:\texttt{NInd}\ \mathbf{a}_3 \end{array}\right] & \mathbf{f} \\ \Longrightarrow & \mathbf{a}:\mathbf{s} & \mathbf{d} & \mathbf{h}[\mathbf{a}:\texttt{NAp}\ \mathbf{a}_1\ \mathbf{a}_3] & \mathbf{f} \\ \hline \end{array}$$

In order to bring this rule into play, we need to modify Rule 2.6 so that it unwinds anew from the root of the redex after the evaluation is completed:

$$(2.9)\quad \begin{array}{|lrcll|l|} \hline & \mathbf{a}:\mathbf{a}_1:[] & & \mathbf{d} & \mathbf{h}\left[\begin{array}{l} \mathbf{a}:\texttt{NPrim Neg} \\ \mathbf{a}_1:\texttt{NAp}\ \mathbf{a}\ \mathbf{b} \end{array}\right] & \mathbf{f} \\ \Longrightarrow & \mathbf{b}:[] & (\mathbf{a}_1:[]):\mathbf{d} & \mathbf{h} & & \mathbf{f} \\ \hline \end{array}$$

This is rather tiresome; the implementations developed in subsequent chapters will do a better job.

Exercise 2.15. Write the state transition rules for addition. (The other dyadic arithmetic operations are practically identical.)

2.6.2 Implementing arithmetic

To implement arithmetic we need to make a number of changes. First, we need to redefine the type `tiDump` to be a stack of stacks, whose initial value is empty.

```
> tiDump == [tiStack]
> initialTiDump = []
```

Next, we need to add a new kind of heap node: `NPrim n p` represents a primitive whose name is `n` and whose value is `p`, where `p` is of type `primitive`. As in the case of `NSupercomb` nodes, the name is present in the `NPrim` node solely for debugging and documentation reasons.

```
> node ::= NAp addr addr                        || Application
>          | NSupercomb name [name] coreExpr   || Supercombinator
>          | NNum num                           || Number
>          | NInd addr                          || Indirection
>          | NPrim name primitive               || Primitive
```

As usual, `showNode` needs to be augmented as well, to display `NPrim` nodes. The transition rules given in the previous section suggest that the data type `primitive` should be defined like this:

```
> primitive ::= Neg | Add | Sub | Mul | Div
```

with one constructor for each desired primitive.

Now, just as we needed to allocate an `NSupercomb` node in the initial heap for each supercombinator, so we need to allocate an `NPrim` node in the initial heap for each primitive. Then we can add extra bindings to the **globals** component of the machine state, which map the name of each primitive to the address of its node, just as we did for supercombinators. We can do this easily by modifying the definition of `buildInitialHeap`, like this:

```
> buildInitialHeap :: [coreScDefn] -> (tiHeap, tiGlobals)
> buildInitialHeap sc_defs
>  = (heap2, sc_addrs ++ prim_addrs)
>    where
>    (heap1, sc_addrs)   = mapAccuml allocateSc hInitial sc_defs
>    (heap2, prim_addrs) = mapAccuml allocatePrim heap1 primitives
```

We define an association list giving the mapping from variable names to primitives, thus:

```
> primitives :: assoc name primitive
> primitives = [ ("negate", Neg),
>                ("+", Add),   ("-", Sub),
>                ("*", Mul),   ("/", Div)
>              ]
```

To add further primitives, just add more constructors to the `primitive` type, and more elements to the `primitives` association list.

We can then define `allocatePrim`, very much as we defined `allocateSc`:

```
> allocatePrim :: tiHeap -> (name, primitive) -> (tiHeap, (name, addr))
> allocatePrim heap (name, prim)
>  = (heap', (name, addr))
>    where
>    (heap', addr) = hAlloc heap (NPrim name prim)
```

Next, we need to augment the `dispatch` function in `step` to call `primStep` when it finds a `NPrim` node. `primStep` performs case analysis on the primitive being used, and then calls one of a family of auxiliary functions, `primNeg`, `primAdd` and so on, which actually perform the operation. For the present we content ourselves with negation.

```
> primStep state Neg   = primNeg state
```

`primNeg` needs to do the following:

- Use `getArgs` to extract the address of the argument from the stack, and `hLookup` to get the node pointed to by this address.
- Use the auxiliary function `isDataNode` to check if the argument node is evaluated.
- If it is not evaluated, use Rule 2.9 to set up the new state ready to evaluate the argument. This involves pushing the current stack on the dump, and making a new stack whose only element is the argument to `negate`.
- If it is evaluated, use `hUpdate` to overwrite the root of the redex with an `NNum` node containing the result, and return, having modified the stack appropriately.

Next, we need to implement the new rules for unwinding and for numbers. The definition of `numStep` must be changed to implement Rule 2.7. If the stack contains just one item, the address of an `NNum node`, and the dump is non-empty, `numStep` should pop the top element of the dump and make it into the new stack. If these conditions do not apply, it should signal an error. Similarly, the definition of `apStep` must be changed to implement Rule 2.8. It can do this by checking for an indirection in the argument, and using `hUpdate` to update the heap if so.

Lastly, we need to make a change to `tiFinal`. At present it halts execution when the stack contains a single `NNum`; *but it must now only do this if the dump is empty*, otherwise the new Rule 2.7 will never get a chance to execute!

Exercise 2.16. Implement all these changes to add negation, and test some programs involving negation. For example,

```
main = negate 3
```

or

```
main = twice negate 3
```

You should also test the following program, to show that the handling of indirections is working:

```
main = negate (I 3)
```

The obvious extension now is to implement addition, subtraction and the other arithmetic primitives. If we rush ahead blindly we will find that all these dyadic arithmetic primitives have a rather stereotyped form; indeed they are identical except for the fact that at one point we use `*` or `/` rather than `+`.

To avoid this duplication, we can instead define a single generic function `primArith` and pass to it the required operation as an argument, thus:

```
> primStep state Add = primArith state (+)
> primStep state Sub = primArith state (-)
> primStep state Mul = primArith state (*)
> primStep state Div = primArith state (/)


> primArith :: tiState -> (num -> num -> num) -> tiState
```

This is a simple example of the way in which higher-order functions can enable us to make programs more modular.

Exercise 2.17. Implement `primArith`, and test your implementation.

2.7 Mark 5: Structured data

In this section we will add structured data types to our reduction machine. It would be nice to give an implementation for the `case` expressions of our core language, but it turns out that it is rather hard to do so within the framework of a template instantiation machine. (Our later implementations will not have this problem.) Instead we will use a collection of built-in functions, such as `if`, `casePair` and `caseList`, which allow us to manipulate certain structured types. The template machine will remain unable to handle general structured objects.

Exercise 2.18. Why is it hard to introduce `case` expressions into the template instantiation machine? (Hint: think about what `instantiate` would do with a `case` expression.)

2.7.1 Building structured data

Structured data is built with the family of constructors `Pack{`**t**`,`**a**`}` where **t** gives the tag of the constructor, and **a** gives its arity (Section 1.1.4), so we need a representation for these constructor functions in the graph. They are really a new form of primitive, so we can do this by adding a new constructor `PrimConstr` to the `primitive` type. Now in the equation for `instantiateConstr`, we can instantiate an expression `EConstr` **t a** to the heap node `NPrim "Pack"` (`PrimConstr` **t a**).

Next the question arises of how this primitive is implemented. We need to add a case to `primStep` to match the `PrimConstr` constructor, which calls a new auxiliary function `primConstr`. This should check that it is given enough arguments, and if so build a structured data object in the heap.

To do this we need to add a new constructor, `NData`, to the `node` type to represent structured data objects. The `NData` constructor contains the *tag* of the object, and its *components*.

```
> node ::= NAp addr addr                    || Application
>        | NSupercomb name [name] coreExpr  || Supercombinator
>        | NNum num                         || Number
>        | NInd addr                        || Indirection
>        | NPrim name primitive             || Primitive
>        | NData num [addr]                 || Tag, list of components
```

We can now give the rule for NPrim (PrimConstr $\mathbf{t}$ $\mathbf{n}$):

$$(2.10)\quad \begin{array}{|lrcc|c|} \hline & \mathbf{a}:\mathbf{a_1}:\ldots:\mathbf{a_n}:[] & \mathbf{d} & \mathbf{h}\left[\begin{array}{l}\mathbf{a}:\texttt{NPrim (PrimConstr } \mathbf{t}\ \mathbf{n}\texttt{)}\\ \mathbf{a_1}:\texttt{NAp}\ \mathbf{a}\ \mathbf{b_1}\\ \ldots\\ \mathbf{a_n}:\texttt{NAp}\ \mathbf{a_{n-1}}\ \mathbf{b_n}\end{array}\right] & \mathbf{f} \\ \Longrightarrow & \mathbf{a_n}:[] & \mathbf{d} & \mathbf{h}[\mathbf{a_n}:\texttt{NData}\ \mathbf{t}\ [\mathbf{b_1},\ldots,\mathbf{b_n}]] & \mathbf{f} \\ \hline \end{array}$$

So much for building structured objects. The next question is how to take them apart, which is expressed by `case` expressions in the Core language. As already mentioned, it is hard to implement `case` expressions directly, so we content ourselves with a few special cases, beginning with booleans.

2.7.2 Conditionals

The boolean type might be declared in Miranda like this:

```
boolean ::= False | True
```

There are two constructors, `True` and `False`. Each has arity zero, and we arbitrarily assign a tag of one to `False` and two to `True`. So we can give the following Core-language definitions:

```
False = Pack{1,0}
True = Pack{2,0}
```

Since we cannot have general `case` expressions, it will suffice to add a conditional primitive, with the reduction rules:

```
if Pack{2,0} t e = t
if Pack{1,0} t e = e
```

Operationally, `if` evaluates its first argument, which it expects to be a data object, examines its tag, and selects either its second or third argument depending on whether the tag is 2 (`True`) or 1 (`False`) respectively.

Exercise 2.19. Write the state transition rules for the conditional primitive. You need three rules: two to perform the reduction if the boolean condition is already evaluated; and one to start evaluation if the condition is not evaluated, by pushing the old stack on the dump, and pushing the address of the condition on the new empty stack (cf. Rule 2.9). You should find that you need to use an indirection in the update for the first two rules.

One further rule is missing. What is it? (Hint: when evaluation of the condition is complete, how does the conditional get re-tried?)

Once you have `if`, you can give Core-language definitions for the other boolean operators in terms of it and `False` and `True`. For example:

```
and x y = if x y False
```

Exercise 2.20. Give Core-language definitions for `or`, `xor` and `not`. Add all of these Core-language definitions to `extraPreludeDefs`.

Finally, we need some way of comparing numeric values, which requires new primitives `>`, `>=` and so on.

2.7.3 Implementing structured data

Here is a list of the changes required to the implementation to add structured data objects, conditionals and comparison operations.

- Add the `NData` constructor to the `node` data type. Extend `showNode` to display `NData` nodes.
- Add `PrimConstr`, `If`, `Greater`, `GreaterEq`, `Less`, `LessEq`, `Eq`, `NotEq` to the `primitive` type. For all except the first, add suitable pairs to the `primitives` association list, so that the names of these primitives can be mapped to their values by `instantiateVar`.
- Add a definition for `instantiateConstr` (and `instantiateAndUpdateConstr` if necessary).
- The `isDataNode` function should identify `NData` nodes as well as `NNum` nodes.
- The `dispatch` code in `step` needs an extra case for `NData` nodes, calling a new auxiliary function `dataStep`.
- Define `dataStep`; it is very similar to `numStep`.
- Extend `primStep` to cope with the new primitives `PrimConstr`, `If`, `Greater` and so on. For `PrimConstr` and `If` it should call new auxiliary functions `primConstr` and `primIf`. The comparison primitives can almost, but not quite, use `primArith`. What we need is a slight generalisation of `primArith`:

  ```
  > primDyadic :: tiState -> (node -> node -> node) -> tiState
  ```

  which takes a node-combining function instead of a number-combining one. It is simple to define `primArith`, and a similar function `primComp` for comparison primitives, in terms of `primDyadic`; and to define `primDyadic` by generalising the definition of `primArith`.

Exercise 2.21. Make all these changes. Now, at last, we can write sensible recursive functions, because we have a conditional to terminate the recursion. Try, for example, the factorial function

```
fac n = if (n == 0) 1 (n * fac (n-1)) ;
main = fac 3
```

2.7.4 Pairs

The constructors for booleans both have arity zero. Next, we will add the data type of pairs, which might be declared in Miranda like this:

```
pair * ** ::= MkPair * **
```

We can build pairs using the Pack{1,2} constructor:

```
MkPair = Pack{1,2}
```

How about taking them apart, still without using case expressions? For example, consider the following Core-language program involving a case expression:

```
f p = case p of
        <1> a b -> b*a*a end
```

Lacking case expressions, we can translate it instead as follows:

```
f p = casePair p f'
f' a b = b*a*a
```

Here, f' is an auxiliary function, and casePair is a built-in primitive defined like this:

```
casePair (Pack{1,2} a b) f = f a b
```

Operationally, casePair evaluates its first argument, which it expects to yield a pair; it then applies its second argument to the two components of the pair. You can implement this by adding yet another constructor PrimCasePair to the primitive type, and writing some more code to handle it.

We can, for example, define fst and snd which extract the first and second components of a pair, with the following Core-language definitions:

```
fst p = casePair p K
snd p = casePair p K1
```

Exercise 2.22. Write the state transition rules for casePair. As usual, you will need two rules: one to perform the reduction if the first argument is evaluated, and one to start its evaluation if not.

Make the necessary changes to implement pairs, as described above.

Test your implementation with the following program (and others of your own):

```
main = fst (snd (fst (MkPair (MkPair 1 (MkPair 2 3)) 4)))
```

2.7.5 Lists

Now that you have done pairs and booleans, lists should be easy. The list data type might be defined in Miranda like this:

```
list * ::= Nil | Cons * (list *)
```

We assign the tag 1 to `Nil` and 2 to `Cons`.

The only question is what exactly the `caseList` primitive, which takes a list apart, should do. We recall that the `casePair` has one 'continuation', a function which takes the components of the pair as its arguments. `if` has two 'continuations', and selects one or other depending on the value of its first argument. So `caseList` is just a combination of both these ideas:

```
caseList Pack{1,0}        cn cc = cn
caseList (Pack{2,2} x xs) cn cc = cc x xs
```

It takes three arguments, and evaluates the first one. If it is an empty list (i.e. has a tag of 1 and no components) then `caseList` simply selects its second argument `cn`. Otherwise it must be a list cell (i.e. a tag of 2 and two components), and `caseList` applies its third argument `cc` to these components.

For example, suppose we wanted to implement the `length` function, which in Miranda would be written

```
length [] = 0
length (x:xs) = 1 + length xs
```

With the aid of `caseList` we could write length like this:

```
length xs = caseList xs 0 length'
length' x xs = 1 + length xs
```

Exercise 2.23. Write Core-language definitions for `Cons`, `Nil`, `head` and `tail`. To define `head` and `tail`, you will need to introduce a new primitive `abort`, which is returned if you take the `head` or `tail` of an empty list. `abort` can conveniently be implemented by calling Miranda's `error` primitive to stop the program.

Exercise 2.24. Write the state transition rules for `caseList`, implement it and `abort`, and add definitions to `preludeDefs` for `Cons`, `Nil`, `head` and `tail`.

Write some programs to test your implementation.

You should now be able to write a suitable `case` primitive for any structured data type you care to think of.

Exercise 2.25. What is the main disadvantage of taking apart structured data types with `case` primitives, rather than implementing full `case` expressions?

2.7.6 Printing lists

So far we have been implicitly assuming that the result of the whole program is a number. What would we have to do to allow a *list* of numbers to be the result? If this was the case, after evaluating `main` for a while, we would eventually expect to find the address of a list object on top of the stack. If it is an empty list, the program terminates. If it is a `Cons` cell, and its head is not evaluated we need to begin a recursive evaluation of its head; if the head is evaluated we need to print the head, and then repeat the whole exercise on the tail.

As soon as we start to write state transition rules to describe this, we have to decide how to express the idea of 'printing a number' in our state transition world. The neatest solution is to add a new component to the state, called the *output*, and model 'printing a number' by 'appending a number to the output'. We will also add two new primitives, `Print` and `Stop`.

The `Stop` primitive is easy: it makes the stack empty. (`tiFinal` will be altered to stop the machine when it sees an empty stack, rather than giving an error, which is what it does now.) `Stop` expects the dump to be empty.

$$\text{(2.11)}\quad \begin{array}{|llllll|} \hline & \mathbf{o} & \mathbf{a} : [\,] & [\,] & \mathbf{h}[\mathbf{a} : \texttt{NPrim Stop}] & \mathbf{f} \\ \Longrightarrow & \mathbf{o} & [\,] & [\,] & \mathbf{h} & \mathbf{f} \\ \hline \end{array}$$

The `Print` primitive evaluates its first argument to an integer, and attaches its value to the output list; then it returns its second argument as its result. It also expects the dump to be empty. The first rule applies if the first argument is already evaluated:

$$\text{(2.12)}\quad \begin{array}{|llllll|} \hline & \mathbf{o} & \mathbf{a} : \mathbf{a}_1 : \mathbf{a}_2 : [\,] & [\,] & \mathbf{h}\left[\begin{array}{l} \mathbf{a} : \texttt{NPrim Print} \\ \mathbf{a}_1 : \texttt{NAp}\ \mathbf{a}\ \mathbf{b}_1 \\ \mathbf{a}_2 : \texttt{NAp}\ \mathbf{a}_1\ \mathbf{b}_2 \\ \mathbf{b}_1 : \texttt{NNum}\ \mathbf{n} \end{array}\right] & \mathbf{f} \\ \Longrightarrow & \mathbf{o} +\!\!+ [\mathbf{n}] & \mathbf{b}_2 : [\,] & [\,] & \mathbf{h} & \mathbf{f} \\ \hline \end{array}$$

`Print` is a rather weird supercombinator, because it has a side-effect on the output **o**. `Print` must obviously be used with care! The second rule applies if `Print`'s first argument is not evaluated: it starts an evaluation in the usual way.

$$\text{(2.13)}\quad \begin{array}{|llllll|} \hline & \mathbf{o} & \mathbf{a} : \mathbf{a}_1 : \mathbf{a}_2 : [\,] & [\,] & \mathbf{h}\left[\begin{array}{l} \mathbf{a} : \texttt{NPrim Print} \\ \mathbf{a}_1 : \texttt{NAp}\ \mathbf{a}\ \mathbf{b}_1 \\ \mathbf{a}_2 : \texttt{NAp}\ \mathbf{a}_1\ \mathbf{b}_2 \end{array}\right] & \mathbf{f} \\ \Longrightarrow & \mathbf{o} & \mathbf{b}_1 : [\,] & (\mathbf{a}_2 : [\,]) : [\,] & \mathbf{h} & \mathbf{f} \\ \hline \end{array}$$

Now, we define the following extra functions in `extraPreludeDefs`:

```
printList xs = caseList xs stop printCons
printCons h t = print h (printList t)
```

where `"print"` is bound by `primitives` to the `Print` primitive, and `"stop"` is bound to `Stop`.

Finally, we modify the `compile` function so that the stack initially contains the address of the expression `(printList main)`. It should not take long to convince yourself that this does the right thing.

Exercise 2.26. Implement these changes, and test your implementation by writing a program which returns a list of numbers.

2.8 Alternative implementations†

These exercises explore some alternative implementations for things we have done.

2.8.1 An alternative representation for primitives

Since *the only thing we ever do to a primitive is execute it*, we can play the following trick: instead of making `primitive` be an enumeration type on which we perform case analysis, we could make it a *function* which takes a `tiState` to a `tiState`, like this:

```
>  primitive == tiState -> tiState
```

The constructors `Add`, `Sub` and so on have vanished altogether. Now the 'case analysis' done by `primStep` is rather easy: just apply the function! Here are the revised definitions for `primStep` and `primitives`.

```
> primStep state prim = prim state
> primitives = [ ("negate", primNeg),
>                ("+", primArith (+)), ("-", primArith (-)),
>                ("*", primArith (*)), ("/", primArith (/))
>              ]
```

This has a direct counterpart in real implementations: instead of storing a small integer tag in a `NPrim` node to distinguish among primitives, we store a code pointer, and jump to it to execute the primitive.

Exercise 2.27. Implement and test this change.

2.8.2 An alternative representation of the dump

At present we have implemented the dump as a stack of stacks, but in a real implementation we would doubtless build the new stack directly on top of the old one. The dump would then contain offsets from the base of the spine stack, telling where one sub-stack ends and the next begins.

We can model this directly in our machine with the following type declaration:

```
> tiDump == stack num
```

Exercise 2.28. Implement and test this change. You will need to modify the equations that deal with beginning and ending an evaluation of an argument, and the definition of `tiFinal`.

2.8.3 An alternative representation for data values

There is another way to implement booleans which is quite instructive. The reduction rules given in Section 2.7.2 for `if` simply select one or other of the second or third arguments. Now, suppose that instead of representing a boolean value as a structured data object, we represented it as a *function, which selects one or other of its arguments.* That is, `True` and `False` are redefined like this:

```
True  t f = t
False t f = f
```

Now boolean operators can be defined like this:

```
if = I
and b1 b2 t f = b1 (b2 t f) f
or b1 b2 t f = b1 t (b2 t f)
not b t f = b f t
```

These definitions can all be included in `extraPreludeDefs`. Now the only primitives required are the arithmetic comparison operators! There is no need to define a primitive `if`, or to add `NData` to the `node` type.

We can apply exactly the same trick for pairs. A pair is represented as a *function* which takes a single argument and applies it to the two components of the pair:

```
pair a b f = f a b
casePair = I
fst p = p K
snd p = p K1
```

The same trick works for lists, but now we need two 'extra' arguments, one to use if the list is a `Cons` cell, and the other to use if it is empty:

```
cons a b cn cc = cc a b
nil      cn cc = cn
caseList = I
```

Exercise 2.29. Implement booleans, pairs and lists in this way, and measure their performance. What advantages and disadvantages can you see relative to the previous implementation?

2.9 Garbage collection†

As execution proceeds, more and more nodes will be allocated in the heap, so the Miranda data structure representing the heap will become larger and larger. Eventually, Miranda will run out of space. This comes as no surprise, because it corresponds directly to real implementations. As nodes are allocated, the heap becomes larger and larger, and eventually fills up. We need to perform *garbage collection* to free up some space.

More specifically, we need to define a function `gc`, with type

```
> gc :: tiState -> tiState
> gc :: TiState -> TiState
```

whose result state behaves exactly like its input state, except that it has a (hopefully) smaller heap. This smaller heap contains all the nodes which are accessible from the other components of the machine state, directly or indirectly. `gc` makes the heap smaller by calling `hFree` on the addresses of nodes which are no longer required (see Appendix A.1 for a description of `hFree`).

The `doAdmin` function can check the heap size (using `hSize`) after each step, and call the garbage collector if it is larger than some given size.

2.9.1 Mark-scan collection

To begin with, we will develop a *mark-scan collector*. This works in three phases:

1. The first phase identifies all the *roots*; that is, all the heap addresses contained in the machine state. Where can such addresses be lurking? We can easily find out by looking at the types involved in the machine state for occurrences of `addr`. The answer is that addresses can occur in the stack, the dump and the globals. So we need the following functions:

```
> findStackRoots  :: tiStack -> [addr]
> findDumpRoots   :: tiDump -> [addr]
> findGlobalRoots :: tiGlobals -> [addr]
```

2. In the *mark phase*, each node whose address is in the machine state is *marked*. When a node is marked, all its descendants are also marked, and so on recursively. The `markFrom` function takes a heap and an address, and returns a new heap in which all the nodes accessible from the address have been marked.

```
> markFrom :: tiHeap -> addr -> tiHeap
```

3. In the *scan phase*, all the nodes in the heap (whether marked or not) are examined. Unmarked nodes are freed, and marked nodes are unmarked.

```
> scanHeap :: tiHeap -> tiHeap
```

Exercise 2.30. Write a definition for `gc` in terms of `findRoots`, `markFrom` and `scanHeap`, and call it appropriately from `doAdmin`.

Exercise 2.31. Write a definition for `findRoots`.

Before we can implement `markFrom` and `scanHeap` we need to have a way to *mark* a node. In a real implementation this is done by using a bit in the node to indicate whether or not the node is marked. We will model this by adding a new constructor, the `node` type, as follows:

```
> node ::= NAp addr addr                       || Application
>        | NSupercomb name [name] coreExpr     || Supercombinator
>        | NNum num                            || Number
>        | NInd addr                           || Indirection
>        | NPrim name primitive                || Primitive
>        | NData num [addr]                    || Tag, list of components
>        | NMarked node                        || Marked node
```

The new kind of node is an `NMarked` node, and it contains inside it the `node` which was there before the marking happened. The `node` inside an `NMarked` node is never another `NMarked` node.

Now we are ready to define `markFrom`. Given an address **a** and a heap **h**, it does the following:

1. It looks up **a** in **h**, giving a node **n**. If it is already marked, `markFrom` returns immediately. This is what prevents the marking process from going on forever when it encounters a cyclic structure in the heap.
2. It marks the node by using `hUpdate` to replace it with `NMarked` **n**.
3. It extracts any addresses from inside **n** (there may be zero or more such addresses), and calls `markFrom` on each of them.

All that remains is `scanHeap`. It uses `hAddresses` to extract the list of all the addresses used in the heap, and examines each in turn. If the node to which it refers is unmarked (that is, not an `NMarked` node), it calls `hFree` to free the node. Otherwise, it unmarks the node by using `hUpdate` to replace it with the node found inside the `NMarked` constructor.

Exercise 2.32. Write definitions for `markFrom` and `scanHeap`.

That completes the mark-scan garbage collector.

Mark-scan is not the only way to perform garbage collection, and we now suggest some directions for further exploration. A brief survey of garbage-collection techniques can be found in Chapter 17 of [Peyton Jones 1987]; a more comprehensive review is [Cohen 1981].

2.9.2 Eliminating indirections

We begin with an optimisation to the collector we have just developed. During evaluation we may introduce indirection nodes, and it would be nice to eliminate them, by readjusting pointers as suggested in Figure 2.2. To do this, we need to change the functionality of `markFrom` a bit. It should now take an address and a heap, mark all the nodes accessible from the address, and return a new heap *together with a new address which should be used instead of the old one.*

```
> markFrom :: tiHeap -> addr -> (tiHeap, addr)
```

In the picture, calling `markFrom` with the address of node (a) should mark node (c) (but not node (b)), and return the address of node (c).

How do we make use of the address returned by `markFrom`? It must be inserted in place of the address with which `markFrom` was called. The easiest way to do this is to merge the first two phases, so that as each root is identified in the machine state, `markFrom` is called, and the returned address is used to replace the original root in the machine state. So we replace `findStackRoots` and its companions with:

```
> markFromStack   :: tiHeap -> tiStack   -> (tiHeap,tiStack)
> markFromDump    :: tiHeap -> tiDump    -> (tiHeap,tiDump)
> markFromGlobals :: tiHeap -> tiGlobals -> (tiHeap,tiGlobals)
```

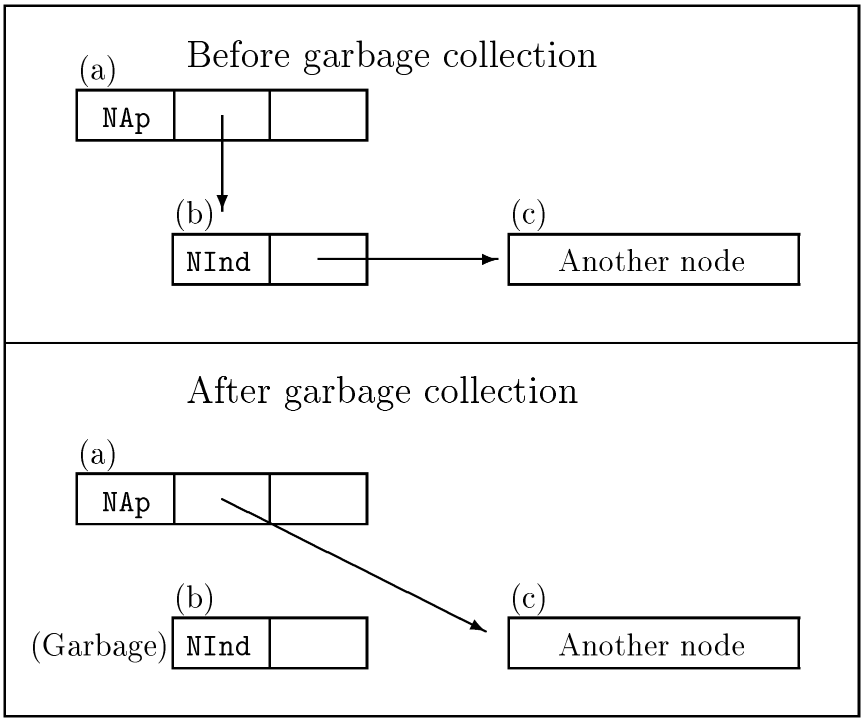


Figure 2.2: Eliminating indirections during garbage collection

Exercise 2.33. Implement the revised version of `markFrom`, making it 'skip over' indirections without marking them, and update the addresses inside each node as it calls itself recursively. Then implement the other marking functions in terms of `markFrom`, and glue them together with a new version of `gc`. Measure the improvement, by comparing the heap sizes obtained with this new collector to the ones you obtained before. (You can easily revert to the one before by removing the special handling of `NInd` from `markFrom`.)

2.9.3 Pointer reversal

If all the **N** nodes in the heap happened to be linked together into a single long list, then `markFrom` would call itself recursively **N** times. In a real implementation this would build up a stack which is as deep as the heap is large. It is very tiresome to have to allocate a stack as large as the heap to account for a very unlikely situation!

There is a neat trick called *pointer reversal*, which can eliminate the stack by linking together the very nodes which are being marked [Schorr and Waite 1967]. The only extra requirement placed by the algorithm is that marked nodes need a few extra bits of state information. We can express this by expanding the `NMarked` constructor somewhat:

```
> node ::= NAp addr addr                       || Application
>        | NSupercomb name [name] coreExpr     || Supercombinator
>        | NNum num                            || Number
>        | NInd addr                           || Indirection
>        | NPrim name primitive                || Primitive
>        | NData num [addr]                    || Tag, list of components
```

```
>           | NMarked markState node            || Marked node
```

```
> markState ::= Done            || Marking on this node finished
>                | Visits num   || Node visited n times so far
```

The meaning of the constructors for `markState` will be explained shortly.

We can describe the pointer-reversal algorithm with the aid of another (quite separate) state transition system. The state of the marking machine has three components $(\mathbf{f}, \mathbf{b}, \mathbf{h})$, the *forward pointer*, the *backward pointer* and the heap. Each call to `markFrom` initiates a new run of the machine. When `markFrom` is called with address $\mathbf{a}$ and heap $\mathbf{h_{init}}$ the machine is started from the state

$$(\mathbf{a}, \mathtt{hNull}, \mathbf{h_{init}})$$

(`hNull` is a distinguished value of type `addr` which does not address any object in the heap, and which can be distinguished from ordinary addresses.) The machine terminates when it is in a state

$$(\mathbf{f},\ \mathtt{hNull},\ \mathbf{h}[\mathbf{f} : \mathtt{NMarked\ Done}\ \mathbf{n}])$$

that is, when $\mathbf{f}$ points to a marked node, and $\mathbf{b} = \mathtt{hNull}$. (It is possible that the initial state is also a final state, if the node pointed to by $\mathbf{f}$ is already marked.)

We begin the transition rules by describing how the machine handles unmarked nodes. `NData` nodes will be ignored for the present. First, we deal with the case of applications. When we encounter an unmarked application, we 'go down' into its first sub-graph, recording the old back-pointer in the first field of the `NAp` node. The new forward-pointer addresses the first sub-graph, and the new back-pointer addresses the application node itself. The state information, `Visits 1`, records the fact that the back-pointer is kept in the first field of the `NAp` node.

	$\mathbf{f}$	$\mathbf{b}$	$\mathbf{h}[\mathbf{f} : \mathtt{NAp}\ \mathbf{a}_1\ \mathbf{a}_2]$
$\Longrightarrow$	$\mathbf{a}_1$	$\mathbf{f}$	$\mathbf{h}[\mathbf{f} : \mathtt{NMarked}\ (\mathtt{Visits\ 1})\ (\mathtt{NAp}\ \mathbf{b}\ \mathbf{a}_2)]$

This is illustrated in Figure 2.3(a) and (b). In this figure the marks are abbreviated to '`V1`' for '`Visits 1`', '`V2`' for '`Visits 2`' and '`D`' for '`Done`'. Notice the way that a chain of reversed pointers builds up in the application nodes which have been visited.

The next rule says that unmarked `NPrim` nodes should be marked as completed, using the `NMarked Done` constructor (Figure 2.3(c)):

	$\mathbf{f}$	$\mathbf{b}$	$\mathbf{h}[\mathbf{f} : \mathtt{NPrim}\ \mathbf{p}]$
$\Longrightarrow$	$\mathbf{f}$	$\mathbf{b}$	$\mathbf{h}[\mathbf{f} : \mathtt{NMarked\ Done}\ (\mathtt{NPrim}\ \mathbf{p})]$

`NSupercomb` and `NNum` nodes are treated similarly, since they do not contain any further addresses.

So much for unmarked nodes. When the machine finds that $\mathbf{f}$ points to a marked node, it inspects the node which $\mathbf{b}$ points to. If it is `hNull`, the machine terminates. Otherwise, it must be a marked `NAp` node. Let us deal first with the case where the state information is `(Visits 1)`, saying that the node has been visited once. We have therefore completed marking

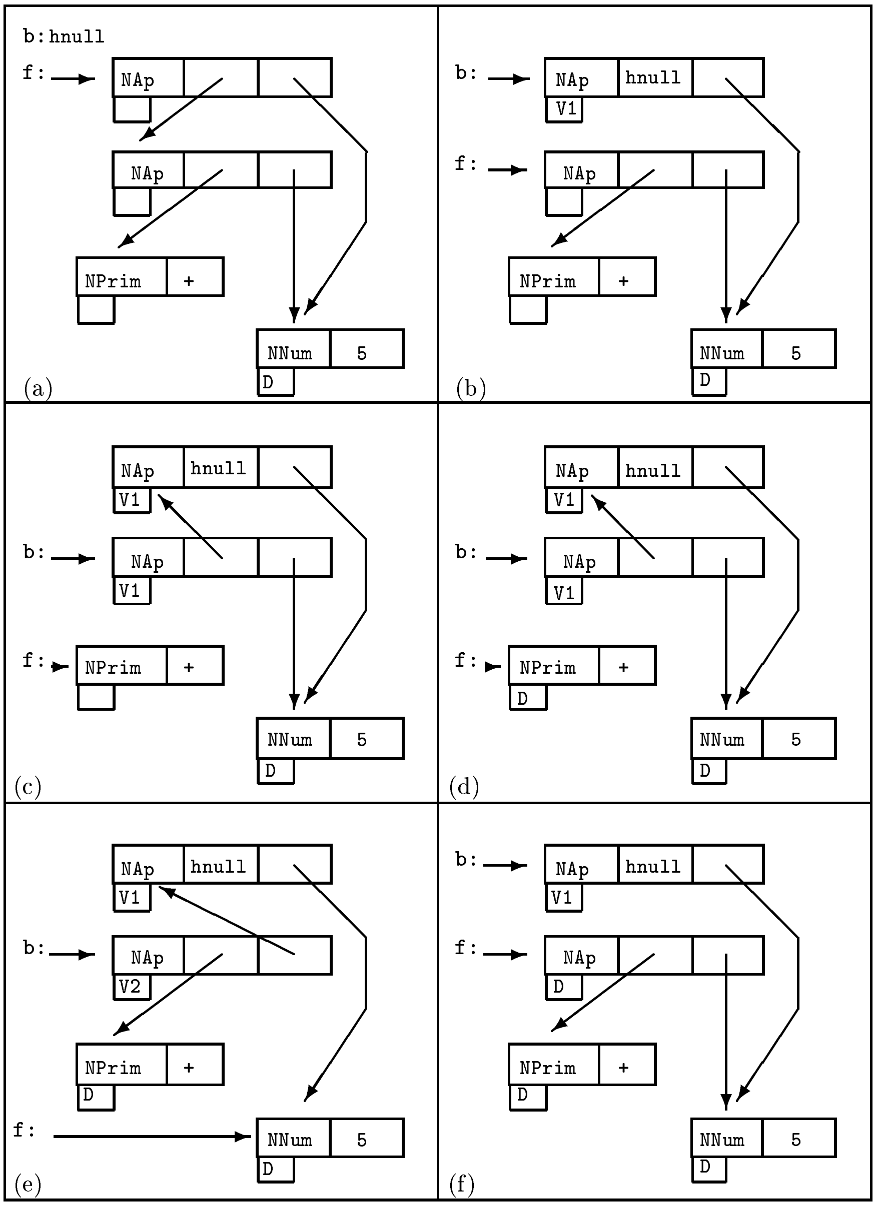


Figure 2.3: Marking a graph using pointer reversal

the first sub-graph of the `NAp` node, and should now mark the second, which we do by making $\mathbf{f}$ point to it, leaving $\mathbf{b}$ unchanged, moving the back-pointer saved in the node ($\mathbf{b}'$) from the first field to the second, and changing the state information (Figure 2.3(d)):

$$\begin{array}{|llll|}\hline & \mathbf{f} & \mathbf{b} & \mathbf{h}\left[\begin{array}{l}\mathbf{f} : \texttt{NMarked Done } \mathbf{n} \\ \mathbf{b} : \texttt{NMarked (Visits 1) (NAp } \mathbf{b}'\ \mathbf{a}_2\texttt{)}\end{array}\right] \\ \Longrightarrow & \mathbf{a}_2 & \mathbf{b} & \mathbf{h}[\mathbf{b} : \texttt{NMarked (Visits 2) (NAp } \mathbf{f}\ \mathbf{b}'\texttt{)}] \\ \hline\end{array}$$

Some time later, the machine will complete marking the second sub-graph, in which case it can restore the node to its original form, and back up the chain of back-pointers one stage (Figure 2.3(e)):

$$\begin{array}{|llll|}\hline & \mathbf{f} & \mathbf{b} & \mathbf{h}\left[\begin{array}{l}\mathbf{f} : \texttt{NMarked Done } \mathbf{n} \\ \mathbf{b} : \texttt{NMarked (Visits 2) (NAp } \mathbf{a}_1\ \mathbf{b}'\texttt{)}\end{array}\right] \\ \Longrightarrow & \mathbf{b} & \mathbf{b}' & \mathbf{h}[\mathbf{b} : \texttt{NMarked Done (NAp } \mathbf{a}_1\ \mathbf{f}\texttt{)}] \\ \hline\end{array}$$

Lastly, we deal with indirections. They are skipped over by changing $\mathbf{f}$ but not $\mathbf{b}$. The heap is left unchanged, so the indirection itself is not marked. When garbage collection is completed, all indirections will therefore be reclaimed. As you can see, 'shorting out' indirections during garbage collection is very easy with this marking algorithm.

$$\begin{array}{|llll|}\hline & \mathbf{f} & \mathbf{b} & \mathbf{h}[\mathbf{f} : \texttt{NInd } \mathbf{a}] \\ \Longrightarrow & \mathbf{a} & \mathbf{b} & \mathbf{h} \\ \hline\end{array}$$

That completes the state transitions for the pointer-reversal algorithm.

Exercise 2.34. Add rules for the `NData` node.

Exercise 2.35. Implement the algorithm. The main changes required are to the `node` type and to the `markFrom` function. `scan` needs to change in a trivial way, because the format of `NMarked` constructors has changed.

2.9.4 Two-space garbage collection

Another very popular way of performing garbage collection is to copy all the live data from one heap to another, the so-called two-space collector invented by [Fenichel and Yochelson 1969] (see also [Baker 1978, Cheney 1970]). The collector works in two stages:

1. All the nodes pointed to by the machine state (stack, dump, etc.) are *evacuated* from the old heap (called *from-space*) into the initially empty new heap (called *to-space*). A node is evacuated by allocating a copy of it in to-space, and overwriting the from-space copy with a *forwarding pointer* containing the to-space address of the new node. Like `markFrom`, the evacuation routine returns the to-space address of the new node, which is used to replace the old address in the machine state.

2. Then all the nodes in to-space are scanned linearly, starting at the first, and each is *scavenged*. A node $\mathbf{n}$ is scavenged by evacuating any nodes to which it points, replacing their addresses in $\mathbf{n}$ with their new to-space addresses. Scanning stops when the scanning pointer catches up with the allocation pointer.

To implement this, we have to add yet another variant of the `node` type, this time with an `NForward` constructor, which contains a single adddress (the to-space address). (`NMarked` is not needed for this collector.) Instead of `markFromStack` we need `evacuateStack` with type:

```
> evacuateStack :: tiHeap -> tiHeap -> tiStack -> (tiHeap, tiStack)
```

The call (`evacuateStack` **fromheap toheap stk**) evacuates all the nodes in **fromheap** referred to from **stk** into **toheap**, returning the new **toheap** and the new **stk**. Similar functions are required for the dump and globals.

Lastly, we need a function

```
> scavengeHeap :: tiHeap -> tiHeap -> tiHeap
```

where the call (`scavengeHeap` **fromheap toheap**) scavenges nodes in **toheap**, evacuating when necessary nodes from **fromheap** into **toheap**.

Exercise 2.36. Implement this garbage collector.

Chapter 3

The G-machine

In this chapter we introduce our first compiler-based implementation, the G-machine, which was developed at the Chalmers Institute of Technology, Göteborg, Sweden, by Augustsson and Johnsson. The material in this chapter is based on their series of papers [Augustsson 1984, Johnsson 1984] culminating in their Ph.D. theses [Augustsson 1987, Johnsson 1987].

3.1 Introduction to the G-machine

The fundamental operation of the template instantiation machine was to construct an instance of a supercombinator body, implemented by the `instantiate` function. This is a rather slow operation, because `instantiate` must recursively traverse the template *each time an instantiation is performed.* When we think of the machine instructions that are executed by `instantiate`, we see that they will be of two kinds: those concerned with traversing the template, and those concerned with actually constructing the instance.

The 'Big Idea' of the G-machine, and other compiled implementations, is this:

> *Before running the program, translate each supercombinator body to a sequence of instructions which, when executed, will construct an instance of the supercombinator body.*

Executing this code should be faster than calling an instantiation function, because all the instructions are concerned with constructing the instance. There are no instructions required to traverse the template, because all that has been done during the translation process. Running a program is thereby split into two stages. In the first stage a compiler is used to produce some intermediate form of the program; this is referred to as *compile-time.* In the second stage the intermediate form is executed; this is called *run-time.*

Since all we ever do to a supercombinator is to instantiate it, we can discard the original supercombinators once the translation is done, keeping only the compiled code.

In principle, then, we use a *G-machine compiler* to turn a program in our source language into a sequence of *machine language instructions.* Because we may wish to implement our language on many different pieces of hardware (68000 based, or VAX, etc.) it is useful to have an *abstract* *machine*. A good abstract machine has two properties: firstly, it can be easily translated into any concrete machine code (for example 68000 assembler); secondly, it is easy to generate the abstract machine code from the source.

Notice that we are faced with a trade-off here. We can ideally satisfy the first property (easy concrete code generation) by making the abstract machine the *same* as the real machine. But this makes the second property much harder to fulfil. An abstract machine is therefore a stepping-stone between the source language and a particular machine code.

3.1.1 An example

Here is a small example of the G-machine compiler in action. Consider the function

```
f g x = K (g x)
```

This would be compiled to the sequence of G-code instructions:

```
Push 1
Push 1
Mkap
Pushglobal K
Mkap
Slide 3
Unwind
```

In Figure 3.1, we show how this code will execute. On the left-hand side of each diagram is the stack, which grows downwards. The remainder of each diagram is the heap. The application nodes are represented by an @ character, expressions are labelled with lower-case letters, and supercombinators are labelled with upper-case letters.

In Figure 3.1, diagram (a), we see the state of the machine before executing the sequence of instructions for `f`. The spine has been unwound, just as it was in the template machine. The top two items on the stack are pointers to the application nodes, whose right-hand parts are the expressions to be bound for `g` and `x`.

The `Push` instruction uses addressing relative to the top of the stack. Ignoring the pointer to the supercombinator node `f`, the first stack item is numbered `0`, the next is numbered `1` and so on. The next diagram (b) shows the changed stack, after executing a `Push 1` instruction. This pushes a pointer to the expression `x` onto the stack, `x` being two stack items down the stack. After another `Push 1` we have a pointer to `g` on top of the stack; again this is two stack items down the stack, because the previous instruction pushed a new pointer onto the stack. The new diagram is (c).

Diagram (d) shows what happens when a `Mkap` instruction is executed. It takes two pointers from the stack and makes an application node from them; leaving a pointer to the result on the stack. In diagram (e) we execute a `Pushglobal K` instruction, with the effect of pushing a pointer to the `K` supercombinator. Another `Mkap` instruction completes the instantiation of the body of `f`, as shown in diagram (f).

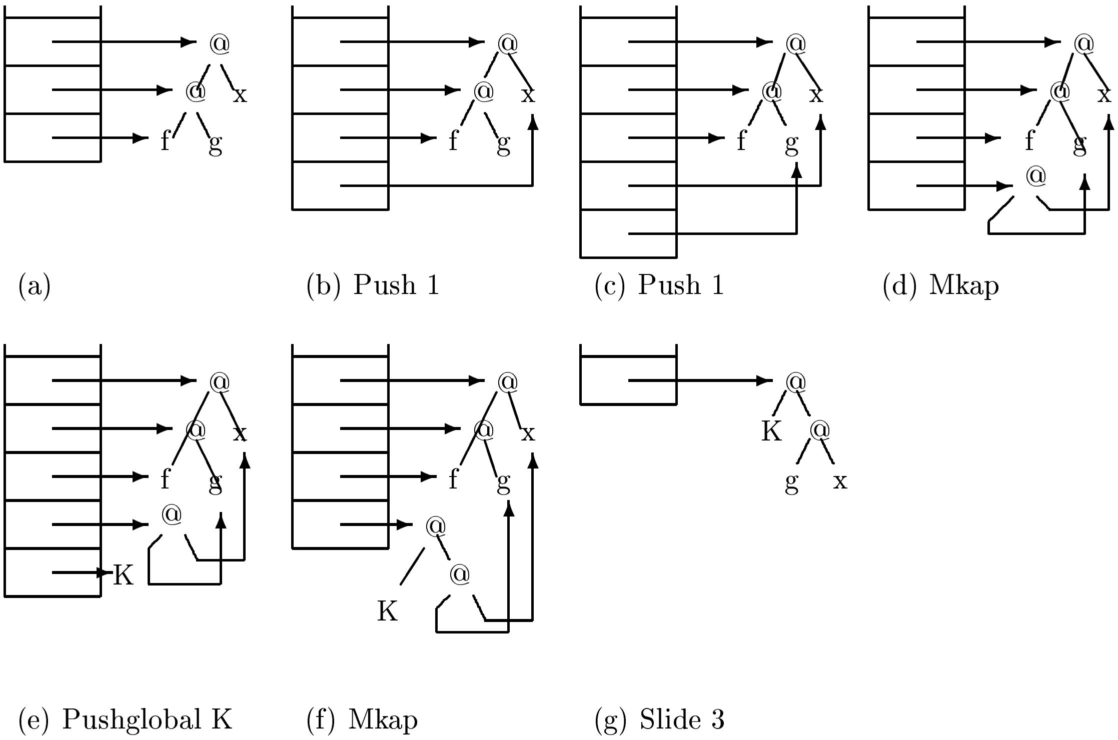


Figure 3.1: Execution of code for the `f` supercombinator

We can now replace the original expression, `f g x`, with the newly instantiated body: `K (g x)`. In the first version of the G-machine – which is not lazy – we simply slide the body down three places on the stack, discarding the three pointers that were there. This is achieved by using a `Slide 3` instruction, as shown in diagram (g). The final `Unwind` instruction will cause the machine to continue to evaluate.

This concludes a brief overview of the execution of the G-machine.

3.1.2 Further optimisations

A modest performance gain can be achieved by eliminating the interpretive overhead of traversing the template, as we have discussed. However, it turns out that compilation also opens the door to a whole host of short-cuts and optimisations which are simply not available to the template instantiation machine. For example, consider the following definition:

```
f x = x + x
```

The template machine would evaluate `x` twice; on the second occasion it would of course find that it was already evaluated. A compiled implementation can spot at compile-time that `x` will already be evaluated, and omit the evaluation step.

3.2 Code sequences for building templates

We recall that the template instantiator operates in the following way:

- The machine has *terminated* when the single item on top of the stack is a pointer to an integer.
- If this is not the case then we *unwind* any application nodes we come across until we reach a supercombinator node. We then *instantiate* a copy of the supercombinator body, making substitutions for its arguments.

At the heart of the Mark 1 template machine are the two functions `scStep` and `instantiate`, which are defined on pages 58 and 59. If we take a look at the definitions of `scStep` and `instantiate`, we can give the following description to the operation of instantiating a supercombinator:

1. Construct a *local environment* of variable names to addresses in the heap.
2. Using this local environment, make an instance of the supercombinator body in the heap. Variables are not copied; instead the corresponding address is used.
3. Remove the pointers to the application nodes and the supercombinator node from the stack.
4. Push the address of the newly created instance of the supercombinator onto the stack.

In the template instantiator, making an instance of a supercombinator involves traversing the tree structure of the expression which is the body of the supercombinator. Because expressions are defined recursively, the tree-traversal function `instantiate` is defined recursively. For example, look at the definition of `instantiate` – on page 59 – for the case of `EAp e1 e2`. First we call `instantiate` for `e1` and then for `e2`, holding on to the addresses of the graph for each sub-expression. Finally we combine the two addresses by building an application node in the graph.

We would like to compile a *linear sequence of instructions* to perform the operation of instantiating an expression.

3.2.1 Postfix evaluation of arithmetic

The desire to construct a linear sequence of instructions to instantiate an expression is reminiscent of the postfix evaluation of arithmetic expressions. We explore this analogy further before returning to the G-machine.

The language of arithmetic expressions consists of: numbers, addition and multiplication. We can represent this language as the type `aExpr`.

```
> aExpr ::= Num num | Plus aExpr aExpr | Mult aExpr aExpr
```

It is intended that the language should have an 'obvious' meaning; we can give this using the function `aInterpret`.

```
> aInterpret :: aExpr -> num
> aInterpret (Num n)      = n
> aInterpret (Plus e1 e2) = aInterpret e1 + aInterpret e2
> aInterpret (Mult e1 e2) = aInterpret e1 * aInterpret e2
```

Alternatively, we can compile the expression into a postfix sequence of operators (or instructions). To evaluate the expression we use the compiled operators and a stack of values. For example, the arithmetic expression $2 + 3 \times 4$ would be represented as the sequence

$$[\texttt{INum}\ 2,\ \texttt{INum}\ 3,\ \texttt{INum}\ 4,\ \texttt{IMult},\ \texttt{IPlus}]$$

We can give the instructions for our postfix machine as the type `aInstruction`.

```
> aInstruction ::= INum num | IPlus | IMult
```

The state of the evaluator is a pair, which is a sequence of operators and a stack of numbers. The meaning of a code sequence is then given in the following transition rules.

$$(3.1)\qquad \boxed{\begin{array}{lrr} & [\,] & [\mathbf{n}] \\ \Longrightarrow & & \mathbf{n} \end{array}}$$

$$(3.2)\qquad \boxed{\begin{array}{lrr} & \texttt{INum}\ \mathbf{n} : \mathbf{i} & \mathbf{ns} \\ \Longrightarrow & \mathbf{i} & \mathbf{n} : \mathbf{ns} \end{array}}$$

$$(3.3)\qquad \boxed{\begin{array}{lrr} & \texttt{IPlus} : \mathbf{i} & \mathbf{n}_0 : \mathbf{n}_1 : \mathbf{ns} \\ \Longrightarrow & \mathbf{i} & (\mathbf{n}_1 + \mathbf{n}_0) : \mathbf{ns} \end{array}}$$

$$(3.4)\qquad \boxed{\begin{array}{lrr} & \texttt{IMult} : \mathbf{i} & \mathbf{n}_0 : \mathbf{n}_1 : \mathbf{ns} \\ \Longrightarrow & \mathbf{i} & (\mathbf{n}_1 \times \mathbf{n}_0) : \mathbf{ns} \end{array}}$$

Translating these transition rules into Miranda gives:

```
> aEval :: ([aInstruction], [num]) -> num
> aEval ([],         [n])     = n
> aEval (INum n:is, s)        = aEval (is, n:    s)
> aEval (IPlus: is, n0:n1:s) = aEval (is, n1+n0:s)
> aEval (IMult: is, n0:n1:s) = aEval (is, n1*n0:s)
```

To generate the sequence of postfix code for an expression we must define a compiler. This takes an expression and delivers a sequence of instructions, which when executed will compute the value of the expression.

```
> aCompile :: aExpr -> [aInstruction]
> aCompile (Num n)      = [INum n]
> aCompile (Plus e1 e2) = aCompile e1 ++ aCompile e2 ++ [IPlus]
> aCompile (Mult e1 e2) = aCompile e1 ++ aCompile e2 ++ [IMult]
```

The key idea from this is given by the type of the `aCompile` function. It returns a list of instructions.

> *The postfix representation of expressions is a way of* flattening *or* linearising *an expression tree, so that the expression can be represented by a flat sequence of operators.*

Exercise 3.1. Using structural induction, or otherwise, prove that the postfix evaluation of arithmetic expressions results in the same answer as the tree evaluation of expressions. That is: prove that for all expressions `e` of type `aExpr`,

$$\texttt{aInterpret e} = \texttt{aEval}\ (\texttt{aCompile e},\ [])$$

This is an example of a *congruence proof.*

Exercise 3.2. Extend the functions `aInterpret`, `aCompile` and `aEval` to handle `let` expressions. Prove that for all expressions in `e` of type `aExpr`, these new functions satisfy the relation:

$$\texttt{aInterpret e} = \texttt{aEval}\ (\texttt{aCompile e},\ [])$$

Can you extend the language to even more complicated expressions, e.g. `letrec` expressions? Can you prove that you have correctly implemented these extensions?

3.2.2 Using postfix code to construct graphs

We can use the same technique to create an instance of a supercombinator body. In this case the 'values' on the stack will be addresses of parts of the expression being instantiated.

The operations of the template construction instructions will be different from those we saw in the arithmetic example above, in that the instructions generally have the side-effect of allocating nodes in the heap. As an example, consider introducing an `Mkap` instruction. This instruction makes an application node, in the heap, from the top two addresses on the stack. It leaves a pointer to this new node on the stack upon completion.

There is no reason to invent a new evaluation stack of addresses, as our template instantiation machine already has such a stack. However, there is an important point to remember if we do make use of this stack:

> *The map of the* stack locations *corresponding to variable names will change as we pop and push objects from the stack. We must therefore keep track of this when we are compiling expressions.*

Our access to items in the stack is relative to the top of the stack. So, if an item is added, the offset to reach that item is increased by one; similarly, when an item is popped, the offset is decreased by one.

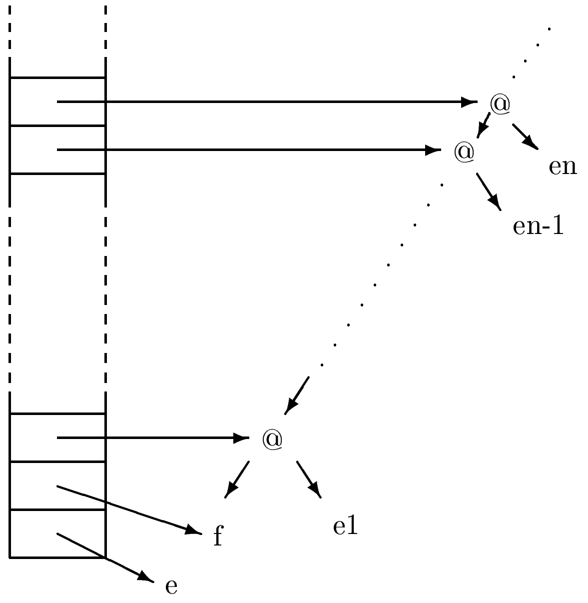


Figure 3.2: The stack layout for the Mark 1 machine

3.2.3 What happens after an instantiation has been made?

Once the instantiation of the supercombinator body has been made we must tidy up the stack, and arrange the continuation of the evaluation process. On completing the evaluation of the postfix sequence for a supercombinator with **n** arguments, the stack will have the following form:

- On top there will be the address in heap of the newly instantiated body, `e`.
- Next there are the $\mathbf{n}+1$ pointers. From these we can access the arguments used in the instantiation process.
- The last of the $\mathbf{n}+1$ pointers points to the root of the expression we have just instantiated.

This is shown in Figure 3.2.

We must replace the redex with the newly instantiated body, and pop off **n** items from the stack, using the `Slide` instruction. To find the next supercombinator we must now start unwinding again, using the `Unwind` instruction. By adding operations to do the tidying and unwinding to the postfix operator sequence, we have transformed the template instantiator into our Mark 1 G-machine.

The code for the function `f x1 ... xn = e` is:

```
<code to construct an instance of e>
Slide n+1
Unwind
```

3.3 Mark 1: A minimal G-machine

We now present the code for a complete G-machine and its compiler. It does not perform updates (which are introduced in Section 3.4) or arithmetic (which is introduced in Section 3.6).

3.3.1 Overall structure

At the top level the G-machine is very similar to the template instantiator; as usual the whole system is knitted together with a `run` function.

```
> run :: [char] -> [char]
> run = showResults . eval . compile . parse
```

The parser data structures and functions are included because we will need access to them.

```
> %include "language" || parser data types
```

3.3.2 Data type definitions

Fundamental to graph reduction implementation techniques is the graph. We use the `heap` data type, amongst others, from the utilities provided in Appendix A.

```
> %include "utils" || heap data type and other library functions
```

The Mark 1 G-machine uses the five-tuple, `gmState`, as its state. A `gmState` holds all the information that we need during the execution of the compiled program.

```
> gmState == (gmCode,           || Current instruction stream
>             gmStack,          || Current stack
>             gmHeap,           || Heap of nodes
>             gmGlobals,        || Global addresses in heap
>             gmStats)          || Statistics
```

In describing the G-machine, we will make use of state *access functions* to access the components of a state. The advantage of this approach is that when we modify the state to accommodate new components, we may reuse most of the original code we have written. We will use the prefix `get` to denote an access function that gets a component from a state, and the prefix `put` to replace a component in a state.

We consider the type definitions of each of the five components of the state, and their access functions, in turn.

- The instruction stream is of type `gmCode` and is simply a list of `instructions`.

  ```
  > gmCode == [instruction]
  ```

To get convenient access to the code, when the state is later augmented with extra components, we define two functions: `getCode` and `putCode`.

```
> getCode :: gmState -> gmCode
> getCode (i, stack, heap, globals, stats) = i
```

```
> putCode :: gmCode -> gmState -> gmState
> putCode i' (i, stack, heap, globals, stats)
>     = (i', stack, heap, globals, stats)
```

There are only six instructions initially. We will describe these in more detail in subsection 3.3.3.

```
> instruction ::= Unwind
>                 | Pushglobal name
>                 | Pushint num
>                 | Push num
>                 | Mkap
>                 | Slide num
```

- The G-machine stack `gmStack` is a list of addresses in the heap.

```
> gmStack == [addr]
```

To get convenient access to the stack, when the state is later augmented with extra components, we define two functions `getStack` and `putStack`

```
> getStack :: gmState -> gmStack
> getStack (i, stack, heap, globals, stats) = stack
```

```
> putStack :: gmStack -> gmState -> gmState
> putStack stack' (i, stack, heap, globals, stats)
>     = (i, stack', heap, globals, stats)
```

- Just as we did in the case of the template instantiator, we use the heap data structure from `utils` to implement heaps.

```
> gmHeap == heap node
```

Again, to access this component of the state we define access functions.

```
> getHeap :: gmState -> gmHeap
> getHeap (i, stack, heap, globals, stats) = heap
```

```
> putHeap :: gmHeap -> gmState -> gmState
> putHeap heap' (i, stack, heap, globals, stats)
>    = (i, stack, heap', globals, stats)
```

In the minimal G-machine there are only three types of nodes: numbers, NNum; applications, NAp; and globals, NGlobal.

```
> node ::= NNum num             || Numbers
>        | NAp addr addr        || Applications
>        | NGlobal num gmCode   || Globals
```

Number nodes contain the relevant number; application nodes apply the function at the first address to the expression at the second address. The NGlobal node contains the number of arguments that the global expects and the code sequence to be executed when the global has enough arguments. This replaces the NSupercomb nodes of the template instantiator, which held a template instead of the arity and code.

- Because we will later be making a lazy implementation it is important that there is only one node for each global. The address of a global can be determined by looking up its value in the association list gmGlobals. This corresponds to the tiGlobals component of the template machine.

```
> gmGlobals == assoc name addr
```

The access function we use is getGlobals; in the Mark 1 machine, this component is constant so we do not need a corresponding put function.

```
> getGlobals :: gmState -> gmGlobals
> getGlobals (i, stack, heap, globals, stats) = globals
```

- The statistics component of the state is implemented as an abstract data type.

```
> abstype gmStats
> with  statInitial  :: gmStats
>       statIncSteps :: gmStats -> gmStats
>       statGetSteps :: gmStats -> num
```

The implementation of gmStats is now given.

```
> gmStats == num
> statInitial    = 0
> statIncSteps s = s+1
> statGetSteps s = s
```

To access this component we define getStats and putStats:

```
> getStats :: gmState -> gmStats
> getStats (i, stack, heap, globals, stats) = stats

> putStats :: gmStats -> gmState -> gmState
> putStats stats' (i, stack, heap, globals, stats)
>    = (i, stack, heap, globals, stats')
```

3.3.3 The evaluator

The G-machine evaluator, `eval`, is defined to produce a list of states. The first one is the one constructed by the compiler. If there is a last state, then the result of the evaluation will be on the top of the stack component of the last state.

```
> eval :: gmState -> [gmState]
> eval state = state: restStates
>              where
>              restStates = [],             gmFinal state
>                         = eval nextState, otherwise
>              nextState  = doAdmin (step state)
```

The function `doAdmin` uses `statIncSteps` to modify the statistics component of the state.

```
> doAdmin :: gmState -> gmState
> doAdmin s = putStats (statIncSteps (getStats s)) s
```

The important parts of the evaluator are the functions `gmFinal` and `step` which we will now look at.

Testing for a final state

The G-machine interpreter has finished when the code sequence that it is executing is empty. We express this condition in the `gmFinal` function.

```
> gmFinal :: gmState -> bool
> gmFinal s = getCode s = []
```

Taking a step

The `step` function is defined so that it makes a state transition based on the instruction it is executing.

```
> step :: gmState -> gmState
> step state = dispatch i (putCode is state)
>              where (i:is) = getCode state
```

We `dispatch` on the current instruction `i` and replace the current code sequence with the code sequence `is`; this corresponds to advancing the program counter in a real machine.

```
> dispatch :: instruction -> gmState -> gmState
> dispatch (Pushglobal f) = pushglobal f
> dispatch (Pushint n)    = pushint n
> dispatch Mkap           = mkap
```

```
> dispatch (Push n)        = push n
> dispatch (Slide n)       = slide n
> dispatch Unwind          = unwind
```

As we can see, the `dispatch` function simply selects a state transition to execute.

Let us begin by looking at the transition rules for the postfix instructions. There will be one for each syntactic object in `instruction`. We begin with the `Pushglobal` instruction, which uses the `globals` component of the state to find the unique `NGlobal` node in the `heap` that holds the global **f**. If it cannot find one, it prints a suitable error message.

$$(3.5)\quad \boxed{\begin{array}{lrrll} & \text{Pushglobal } \mathbf{f} : \mathbf{i} & \mathbf{s} & \mathbf{h} & \mathbf{m}[\mathbf{f} : \mathbf{a}] \\ \Longrightarrow & \mathbf{i} & \mathbf{a} : \mathbf{s} & \mathbf{h} & \mathbf{m} \end{array}}$$

We implement this rule using the `pushglobal` function.

```
> pushglobal :: name -> gmState -> gmState
> pushglobal f state
>       = putStack (a: getStack state) state
>       where a = aLookup (getGlobals state) f (error ("Undeclared global " ++ f))
```

The remaining transitions are for constructing the body of a supercombinator. The transition for `Pushint` places an integer node into the heap.

$$(3.6)\quad \boxed{\begin{array}{lrrlr} & \text{Pushint } \mathbf{n} : \mathbf{i} & \mathbf{s} & \mathbf{h} & \mathbf{m} \\ \Longrightarrow & \mathbf{i} & \mathbf{a} : \mathbf{s} & \mathbf{h}[\mathbf{a} : \texttt{NNum}\ \mathbf{n}] & \mathbf{m} \end{array}}$$

The corresponding function is `pushint`. The number is placed in the new heap `heap'` with address `a`. We then place the heap and stack back into the state.

```
> pushint :: num -> gmState -> gmState
> pushint n state
>       = putHeap heap' (putStack (a: getStack state) state)
>       where (heap', a) = hAlloc (getHeap state) (NNum n)
```

The `Mkap` instruction uses the two addresses on the top of the stack to construct an application node in the heap. It has the following transition rule.

$$(3.7)\quad \boxed{\begin{array}{lrrlr} & \text{Mkap} : \mathbf{i} & \mathbf{a}_1 : \mathbf{a}_2 : \mathbf{s} & \mathbf{h} & \mathbf{m} \\ \Longrightarrow & \mathbf{i} & \mathbf{a} : \mathbf{s} & \mathbf{h}[\mathbf{a} : \texttt{NAp}\ \mathbf{a}_1\ \mathbf{a}_2] & \mathbf{m} \end{array}}$$

This transition becomes `mkap`. Again `heap'` and `a` are respectively the new heap and the address of the new node.

```
> mkap :: gmState -> gmState
> mkap state
>       = putHeap heap' (putStack (a:as') state)
>      where (heap', a)  = hAlloc (getHeap state) (NAp a1 a2)
>              (a1:a2:as') = getStack state
```

The Push instruction is used to take a copy of an argument which was passed to a function. To do this it has to 'look through' the application node which is pointed to from the stack. We must also remember to skip over the supercombinator node which is on the stack.

$$\begin{array}{|lrrlr|} & \texttt{Push}\ \mathbf{n} : \mathbf{i} & \mathbf{a}_0 : \ldots : \mathbf{a}_{\mathbf{n}+1} : \mathbf{s} & \mathbf{h}[\mathbf{a}_{\mathbf{n}+1} : \texttt{NAp}\ \mathbf{a}_\mathbf{n}\ \mathbf{a}'_\mathbf{n}] & \mathbf{m} \\ \Longrightarrow & \mathbf{i} & \mathbf{a}'_\mathbf{n} : \mathbf{a}_0 : \ldots : \mathbf{a}_{\mathbf{n}+1} : \mathbf{s} & \mathbf{h} & \mathbf{m} \end{array} \tag{3.8}$$

```
> push :: num -> gmState -> gmState
> push n state
>    = putStack (a:as) state
>    where   as = getStack state
>            a  = getArg (hLookup (getHeap state) (as !! (n+1)))
```

This uses the auxiliary function getArg to select the required expression from an application node.

```
> getArg :: node -> addr
> getArg (NAp a1 a2) = a2
```

*Because of the stack structure we have changed the addressing mode of the **Push** instruction from that used in [Peyton Jones 1987].*

Next, the tidying up of the stack, which occurs after a supercombinator has been instantiated and before continuing unwinding, is performed by the Slide instruction.

$$\begin{array}{|lrrrr|} & \texttt{Slide}\ \mathbf{n} : \mathbf{i} & \mathbf{a}_0 : \ldots : \mathbf{a}_\mathbf{n} : \mathbf{s} & \mathbf{h} & \mathbf{m} \\ \Longrightarrow & \mathbf{i} & \mathbf{a}_0 : \mathbf{s} & \mathbf{h} & \mathbf{m} \end{array} \tag{3.9}$$

```
> slide :: num -> gmState -> gmState
> slide n state
>       = putStack (a: drop n as) state
>       where (a:as) = getStack state
```

Unwind is the most complex instruction because it replaces the outer loop of our template instantiator. The Unwind instruction is always the last instruction of a sequence, as we shall see in the next section. The newState constructed depends on the item on top of the stack; this depends on the transition rule that is fired, which also depends on the item on top of the stack.

```
> unwind :: gmState -> gmState
> unwind state
>      = newState (hLookup heap a)
>      where
>              (a:as) = getStack state
>              heap   = getHeap state
```

We first consider the case where there is a number on top of the stack. In this case, we are finished; the G-machine has terminated, and we place [] in the code component to signify this fact.

$$(3.10) \quad \begin{array}{|lrlll|} \hline & [\texttt{Unwind}] & \mathbf{a} : \mathbf{s} & \mathbf{h}[\mathbf{a} : \texttt{NNum}\ \mathbf{n}] & \mathbf{m} \\ \Longrightarrow & [] & \mathbf{a} : \mathbf{s} & \mathbf{h} & \mathbf{m} \\ \hline \end{array}$$

```
>               newState (NNum n)      = state
```

If there is an application node on top of the stack then we must continue to unwind from the next node.

$$(3.11) \quad \begin{array}{|llrll|} \hline & [\texttt{Unwind}] & \mathbf{a} : \mathbf{s} & \mathbf{h}[\mathbf{a} : \texttt{NAp}\ \mathbf{a_1}\ \mathbf{a_2}] & \mathbf{m} \\ \Longrightarrow & [\texttt{Unwind}] & \mathbf{a_1} : \mathbf{a} : \mathbf{s} & \mathbf{h} & \mathbf{m} \\ \hline \end{array}$$

```
>               newState (NAp a1 a2)   = putCode [Unwind] (putStack (a1:a:as) state)
```

The most complicated rule occurs when there is a global node on top of the stack. There are two cases to consider, depending on whether there are enough arguments to reduce the supercombinator application.

Firstly, if there are not enough arguments to reduce the supercombinator application then the program was ill-typed. We will ignore this case for the Mark 1 G-machine. Alternatively, when there are enough arguments, it is possible to reduce the supercombinator, by 'jumping to' the code for the supercombinator. In the transition rule this is expressed by moving the supercombinator code into the code component of the machine.

$$(3.12) \quad \begin{array}{|lrlll|} \hline & [\texttt{Unwind}] & \mathbf{a_0} : \ldots : \mathbf{a_n} : \mathbf{s} & \mathbf{h}[\mathbf{a_0} : \texttt{NGlobal}\ \mathbf{n}\ \mathbf{c}] & \mathbf{m} \\ \Longrightarrow & \mathbf{c} & \mathbf{a_0} : \ldots : \mathbf{a_n} : \mathbf{s} & \mathbf{h} & \mathbf{m} \\ \hline \end{array}$$

```
>              newState (NGlobal n c)  = error "Unwinding with too few arguments",
>                                                            #as < n
>                                    = putCode c state, otherwise
```

We have now seen how the instructions are defined, but we have not seen how to generate the postfix sequences of operators, or instruction sequences as we shall refer to them from now on. This is the subject of the next subsection.

3.3.4 Compiling a program

We describe the compiler using a set of *compilation schemes*. Each supercombinator definition is compiled using the compilation scheme $\mathcal{SC}$. The compiled code generated for each supercombinator is defined in Figure 3.3. Corresponding to the compilation schemes $\mathcal{SC}$, $\mathcal{R}$ and $\mathcal{C}$ are *compiler functions* `compileSc`, `compileR` and `compileC`. We consider each of these in turn.

$\mathcal{SC}[\![\mathbf{d}]\!]$ is the G-machine code for the supercombinator definition $\mathbf{d}$.

$$\mathcal{SC}[\![\mathbf{f}\ \mathbf{x}_1\ \ldots\ \mathbf{x_n}\ =\ \mathbf{e}]\!]\ =\ \mathcal{R}[\![\mathbf{e}]\!]\ [\mathbf{x}_1 \mapsto 0, \ldots, \mathbf{x_n} \mapsto \mathbf{n}-1]\ \mathbf{n}$$

$\mathcal{R}[\![\mathbf{e}]\!]\ \rho\ \mathbf{d}$ generates code which instantiates the expression $\mathbf{e}$ in environment ρ, for a supercombinator of arity $\mathbf{d}$, and then proceeds to unwind the resulting stack.

$$\mathcal{R}[\![\mathbf{e}]\!]\ \rho\ \mathbf{d}\ =\ \mathcal{C}[\![\mathbf{e}]\!]\ \rho \mathbin{+\!\!+} [\texttt{Slide}\ \mathbf{d}+1,\ \texttt{Unwind}]$$

$\mathcal{C}[\![\mathbf{e}]\!]\ \rho$ generates code which constructs the graph of $\mathbf{e}$ in environment ρ, leaving a pointer to it on top of the stack.

$$\begin{array}{rcll}
\mathcal{C}[\![\mathbf{f}]\!]\ \rho & = & [\texttt{Pushglobal}\ \mathbf{f}] & \text{where } \mathbf{f} \text{ is a supercombinator}\\
\mathcal{C}[\![\mathbf{x}]\!]\ \rho & = & [\texttt{Push}\ (\rho\ \mathbf{x})] & \text{where } \mathbf{x} \text{ is a local variable}\\
\mathcal{C}[\![\mathbf{i}]\!]\ \rho & = & [\texttt{Pushint}\ \mathbf{i}] & \\
\mathcal{C}[\![\mathbf{e}_0\ \mathbf{e}_1]\!]\ \rho & = & \mathcal{C}[\![\mathbf{e}_1]\!]\ \rho \mathbin{+\!\!+} \mathcal{C}[\![\mathbf{e}_0]\!]\ \rho^{+1} \mathbin{+\!\!+} [\texttt{Mkap}] & \text{where } \rho^{+\mathbf{n}}\ \mathbf{x} = (\rho\ \mathbf{x}) + \mathbf{n}
\end{array}$$

Figure 3.3: The $\mathcal{SC}$, $\mathcal{R}$ and $\mathcal{C}$ compilation schemes

The `compile` function turns a program into an initial state for the G-machine. The initial code sequence finds the global `main` and then evaluates it. The heap is initialised so that it contains a node for each global declared. `globals` contains the map from global names to the `NGlobal` nodes provided for them.

```
> compile :: coreProgram -> gmState
> compile program
>    = (initialCode, [], heap, globals, statInitial)
>    where (heap, globals) = buildInitialHeap program
```

To construct the initial heap and to provide the map of the global nodes for each global defined we use `buildInitialHeap`. This is just as it was in the template machine.

```
> buildInitialHeap :: coreProgram -> (gmHeap, gmGlobals)
> buildInitialHeap program
>    = mapAccuml allocateSc hInitial compiled
>    where compiled = map compileSc (preludeDefs ++ program) ++
>                     compiledPrimitives
```

The `buildInitialHeap` function uses `mapAccuml` to allocate nodes for each compiled global; the compilation occurring (where necessary) in `compiled`, which has type `[gmCompiledSC]`.

```
> gmCompiledSC == (name, num, gmCode)
```

The function allocateSc allocates a new global for its compiled supercombinator argument, returning the new heap and the address where the global is stored.

```
> allocateSc :: gmHeap -> gmCompiledSC -> (gmHeap, (name, addr))
> allocateSc heap (name, nargs, instns)
>       = (heap', (name, addr))
>       where (heap', addr) = hAlloc heap (NGlobal nargs instns)
```

In the initial state, we want the machine to evaluate the value of the program. We recall that this is just the value of the global main.

```
> initialCode :: gmCode
> initialCode = [Pushglobal "main", Unwind]
```

Each supercombinator is compiled using compileSc, which implements the $\mathcal{SC}$ scheme of Figure 3.3. It returns a triple containing the supercombinator name, the number of arguments the supercombinator needs before it can be reduced, and the code sequence associated with the supercombinator.

```
> compileSc :: (name, [name], coreExpr) -> gmCompiledSC
> compileSc (name, env, body)
>       = (name, length env, compileR body (zip2 env [0..]))
```

This in turn uses compileR, which corresponds to the $\mathcal{R}$ scheme of Figure 3.3.

```
> compileR :: gmCompiler
> compileR e env = compileC e env ++ [Slide (#env + 1), Unwind]
> compileR e env = compileC e env ++ [Slide (#env + 1), Unwind]
```

Each of the compiler schemes has the same type: gmCompiler.

```
> gmCompiler == coreExpr -> gmEnvironment -> gmCode
```

We use the fact that we can represent the map ρ from the compilation scheme as an association list. Not only can we look up the offsets for a variable from this list, but we may also calculate how many arguments there are on the stack. This is used in compileR to find out how many stack elements to squeeze out with a Slide instruction. The list has type gmEnvironment, which is defined as:

```
> gmEnvironment == assoc name num
```

This constructs the instantiation of the supercombinator body using compileC, which corresponds to the $\mathcal{C}$ scheme of Figure 3.3.

```
> compileC :: gmCompiler
> compileC (EVar v)    env = [Push n],       member (aDomain env) v
```

```
>                           = [Pushglobal v], otherwise
>                             where n = aLookup env v (error "Can't happen")
> compileC (ENum n)    env = [Pushint n]
> compileC (EAp e1 e2) env = compileC e2 env ++
>                            compileC e1 (argOffset 1 env) ++
>                            [Mkap]
```

We can change the stack offsets using the function `argOffset`. If `env` implements ρ, then `(argOffset n env)` implements $\rho^{+\mathbf{n}}$.

```
> argOffset :: num -> gmEnvironment -> gmEnvironment
> argOffset n env = [(v, n+m) | (v,m) <- env]
```

An example compilation

Let us look at the compilation of the `K` combinator. When compiling this function, we will begin by evaluating the following expression.

```
  compileSc ("K", ["x", "y"], EVar "x")
```

The first element of the tuple is the name (`K` in this case); the second is the argument list (in this case we have two variables: `x` and `y`); and the third component of the tuple is the body of the supercombinator (which for this example is just the variable `x`).

When we rewrite this expression, we get:

```
  ("K", 2, compileR (EVar "x") [("x", 0), ("y", 1)])
```

The resulting triple consists of the name (`K`), the number of arguments we need to reduce the supercombinator (two in this case), and the code sequence to perform the instantiation. When we rewrite this expression we will generate the code sequence for this supercombinator. Notice that the environment is represented by the expression `[("x", 0), ("y", 1)]`; this tells us that when we instantiate the body, a pointer to `x` will be on top of the argument stack and a pointer to `y` will be immediately below `x` on the stack.

```
  ("K", 2, compileC (EVar "x") [("x", 0), ("y", 1)] ++ [Slide 3, Unwind])
```

The `compileR` function is defined to compile the body using `compileC`, and to add a `Slide` and an `Unwind` instruction at the end.

To compile the body we look up `x` and find that it is on top of the stack. We generate code to make a copy of the top of the stack, using `Push 0`.

```
  ("K", 2, [Push 0, Slide 3, Unwind])
```

Exercise 3.3. Write out the equivalent sequence of transformations for the `S` combinator from the prelude definitions. Recall that `S` is defined as:

```
S f g x = f x (g x)
```

Check the final result by running the compiler and machine with any of the simple programs given in Appendix B. (S is in the standard prelude.)

Primitives

In this minimal G-machine there are no primitives, so there is nothing to implement!

```
> compiledPrimitives :: [gmCompiledSC]
> compiledPrimitives = []
```

3.3.5 Printing the results

Because a number of the state components are abstract data types (and are therefore not directly printable) we must define a pretty-printer for the states that the machine produces. It is also a fact that the output is voluminous and not very informative if it is all displayed at once. The printing is controlled by `showResults`. It produces three pieces of output: the super-combinator code sequences, the state transitions and the final statistics.

```
> showResults :: [gmState] -> [char]
> showResults states
>       = iDisplay (iConcat [
>       iStr "Supercombinator definitions", iNewline,
>       iInterleave iNewline (map (showSC s) (getGlobals s)),
>       iNewline, iNewline, iStr "State transitions", iNewline, iNewline,
>       iLayn (map showState states),
>       iNewline, iNewline,
>       showStats (last states)])
>       where (s:ss) = states
```

Taking each of these in turn, we begin with `showSC`. This finds the code for the supercombinator in the unique global heap node associated with the global, and prints the code sequence using `showInstructions`.

```
> showSC :: gmState -> (name, addr) -> iseq
> showSC s (name, addr)
>       = iConcat [ iStr "Code for ", iStr name, iNewline,
>             showInstructions code, iNewline, iNewline]
>       where (NGlobal arity code) = (hLookup (getHeap s) addr)
```

Then `showInstructions` is used to output a code sequence.

```
> showInstructions :: gmCode -> iseq
> showInstructions is
>       = iConcat [iStr "  Code:{",
>            iIndent (iInterleave iNewline (map showInstruction is)),
>            iStr "}", iNewline]
```

The output for each individual instruction is given by showInstruction.

```
> showInstruction :: instruction -> iseq
> showInstruction Unwind         = iStr  "Unwind"
> showInstruction (Pushglobal f) = (iStr "Pushglobal ") $iAppend (iStr f)
> showInstruction (Push n)       = (iStr "Push ")       $iAppend (iNum n)
> showInstruction (Pushint n)    = (iStr "Pushint ")    $iAppend (iNum n)
> showInstruction Mkap           = iStr  "Mkap"
> showInstruction (Slide n)      = (iStr "Slide ")      $iAppend (iNum n)
```

The next major piece of output is the state transitions; these are individually dealt with using showState.

```
> showState :: gmState -> iseq
> showState s
>    = iConcat [showStack s,         iNewline,
>    showInstructions (getCode s),   iNewline]
```

To correspond with our diagrams, we would like to have the top of stack at the bottom of the printed stack. To this end we reverse the stack.

```
> showStack :: gmState -> iseq
> showStack s
>       = iConcat [iStr " Stack:[",
>            iIndent (iInterleave iNewline
>                         (map (showStackItem s) (reverse (getStack s)))),
>            iStr "]"]
```

Each stack item is displayed using showStackItem. It prints the address stored in the stack and the object in the heap to which it points.

```
> showStackItem :: gmState -> addr -> iseq
> showStackItem s a
>       = iConcat [iStr (showaddr a), iStr ": ",
>            showNode s a (hLookup (getHeap s) a)]
```

The function showNode needs to invert the association list of global names and heap addresses to display the global nodes it comes across.

```
> showNode :: gmState -> addr -> node -> iseq
> showNode s a (NNum n)      = iNum n
> showNode s a (NGlobal n g) = iConcat [iStr "Global ", iStr v]
>     where v = hd [n | (n,b) <- getGlobals s; a=b]
> showNode s a (NAp a1 a2)   = iConcat [iStr "Ap ", iStr (showaddr a1),
>                                       iStr " ",   iStr (showaddr a2)]
```

Finally, we print the accumulated statistics, using showStats.

```
> showStats :: gmState -> iseq
> showStats s
>       = iConcat [ iStr "Steps taken = ", iNum (statGetSteps (getStats s))]
```

This concludes the description of the basic G-machine. We now move on to consider ways to make it more sophisticated.

3.3.6 Improvements to the Mark 1 G-machine

Exercise 3.4. Run the program `main = S K K 3`. How many steps does it take? Why is it different from that obtained for the template machine? Do you think that comparing steps taken is a fair comparison of the machines?

Exercise 3.5. Try running some other programs from Appendix B. Remember, there is no arithmetic in this simple machine.

Exercise 3.6. It is possible to use the same trick we used for `Pushglobal` to implement `Pushint`: for each distinct number we create a unique node in the heap. For example, when we first execute `Pushint 2`, we update `gmGlobals` so that it associates `"2"` with the address in heap of the node `NNum 2`.

In the transition rules, if there is already a global called $\mathbf{n}$, then we can reuse this global node.

$$(3.13)\quad \boxed{\begin{array}{lrrll} & \texttt{Pushint}\ \mathbf{n} : \mathbf{i} & \mathbf{s} & \mathbf{h} & \mathbf{m}[\mathbf{n} : \mathbf{a}] \\ \Longrightarrow & \mathbf{i} & \mathbf{a} : \mathbf{s} & \mathbf{h} & \mathbf{m} \end{array}}$$

Alternatively, when this is not the case, we will create a new node and add it to the global map.

$$(3.14)\quad \boxed{\begin{array}{lrrll} & \texttt{Pushint}\ \mathbf{n} : \mathbf{i} & \mathbf{s} & \mathbf{h} & \mathbf{m} \\ \Longrightarrow & \mathbf{i} & \mathbf{a} : \mathbf{s} & \mathbf{h}[\mathbf{a} : \texttt{NNum}\ \mathbf{n}] & \mathbf{m}[\mathbf{n} : \mathbf{a}] \end{array}}$$

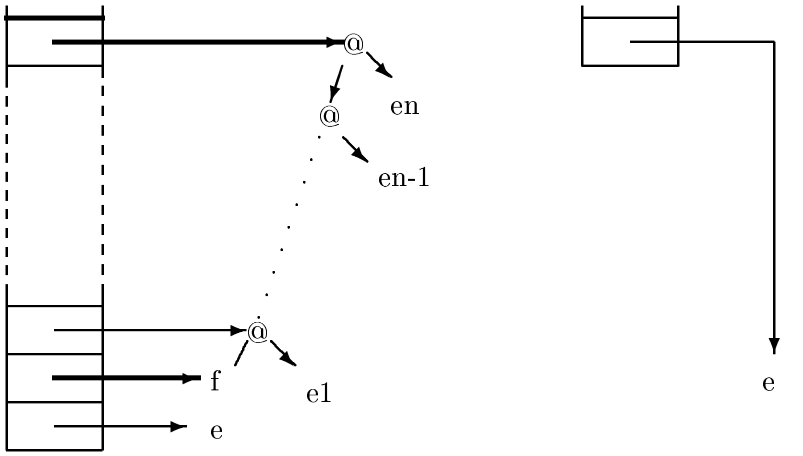
The advantage of this scheme is that we can reuse the same number node in the heap each time a `Pushint` is executed.

Implement this new transition `pushint` for the `Pushint` instruction. You should define an access function for the global component, calling it `putGlobals`.

3.4 Mark 2: Making it lazy

We will now make a number of small changes to the Mark 1 G-machine in order to make it lazy. The Mark 1 machine is not lazy at the moment because it does not overwrite the root node of the original expression before unwinding. This updating is described in Section 2.1.5. In the Mark 2 machine, the idea is that after instantiating the body of the supercombinator, we will *overwrite* the root of the original redex with an *indirection node* pointing to the newly constructed instance. The effect is that the machine 'remembers' the value that was instantiated last time the redex was reduced, and hence does not need to recalculate it.

We implement this change as follows. In the Mark 1 machine the code for each supercombinator concluded with [`Slide` $(\mathbf{n}+1)$, `Unwind`]. To capture updating we replace this with [`Update` $\mathbf{n}$, `Pop` $\mathbf{n}$, `Unwind`]. This is illustrated in the following diagrams, in which we use `#` to represent indirection nodes.



Before executing Slide (n+1) After executing Slide (n+1)

Figure 3.4: Mark 1 G-machine (executing `Slide n+1`)

Figure 3.4 shows how the Mark 1 machine executes a `Slide` $\mathbf{n}+1$ instruction. In Figure 3.5 we see the Mark 2 machine executing the sequence [`Update` **n**, `Pop` **n**]; this being the sequence we propose to use as a lazy replacement for [`Slide` $\mathbf{n}+1$]. The `Update` instruction is responsible for overwriting the root node with the newly created instance of the body of the supercombinator. The `Pop` instruction is used to remove the arguments from the stack, as they are now no longer needed.

Let us first consider the necessary modifications to the data structures.

3.4.1 Data structures

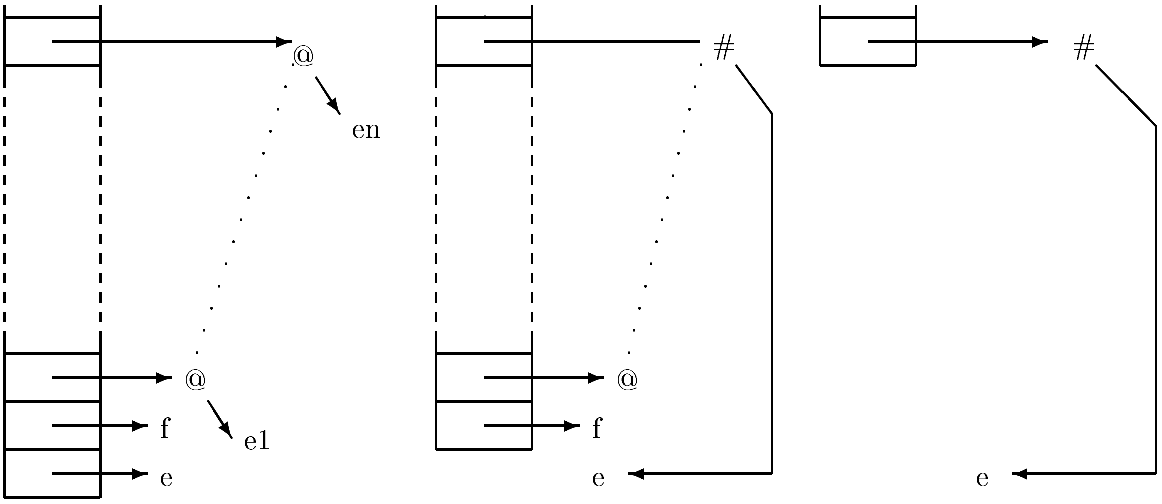
In place of the single instruction `Slide` $\mathbf{n}+1$ that we generated last time we now generate the sequence of instructions [`Update` **n**, `Pop` **n**]. Therefore we are going to have to include these instructions in the new instruction set.

```
> instruction ::= Unwind
>               | Pushglobal name
>               | Pushint num
>               | Push num
>               | Mkap
>               | Update num
>               | Pop num
```

Exercise 3.7. Modify the function `showInstruction`, so that it displays the new instructions.

To implement the indirection nodes we must have a new node type in the heap: `NInd` which we use for indirections.

```
> node ::= NNum num              || Numbers
```



Before executing Update n After Update n After Pop n

Figure 3.5: Mark 2 G-machine (executing `[Update n, Pop n]`)

```
>          | NAp  addr addr      || Applications
>          | NGlobal num gmCode  || Globals
>          | NInd addr           || Indirections
```

Again we must redefine the display function `showNode`, so that it reflects the extension of the data type.

Exercise 3.8. Make the necessary change to `showNode`.

We have not yet given a semantics to the two new instructions. This is done below.

3.4.2 The evaluator

The effect of an `Update` $\mathbf{n}$ instruction is to overwrite the $\mathbf{n}+1^{\text{th}}$ stack item with an indirection to the item on top of the stack. Notice that this addressing mode is different from that used in [Peyton Jones 1987]. For the intended application of this instruction the $\mathbf{a}_1 \ldots \mathbf{a_n}$ are the $\mathbf{n}$ application nodes forming the spine, and $\mathbf{a}_0$ is the function node.

$$(3.15) \quad \boxed{\begin{array}{lrlll} & \texttt{Update}\ \mathbf{n} : \mathbf{i} & \mathbf{a} : \mathbf{a}_0 : \ldots : \mathbf{a_n} : \mathbf{s} & \mathbf{h} & \mathbf{m} \\ \Longrightarrow & \mathbf{i} & \mathbf{a}_0 : \ldots : \mathbf{a_n} : \mathbf{s} & \mathbf{h}[\mathbf{a_n} : \texttt{NInd}\ \mathbf{a}] & \mathbf{m} \end{array}}$$

The `Pop` $\mathbf{n}$ instruction simply removes $\mathbf{n}$ stack items. Again, in the Mark 2 G-machine $\mathbf{a}_1 \ldots \mathbf{a_n}$ are the application nodes forming the spine of the redex.

$$(3.16) \quad \boxed{\begin{array}{lrlll} & \texttt{Pop}\ \mathbf{n} : \mathbf{i} & \mathbf{a}_1 : \ldots : \mathbf{a_n} : \mathbf{s} & \mathbf{h} & \mathbf{m} \\ \Longrightarrow & \mathbf{i} & \mathbf{s} & \mathbf{h} & \mathbf{m} \end{array}}$$

$\mathcal{R}[\![\mathbf{e}]\!]\ \rho\ \mathbf{d}$ generates code which instantiates the expression $\mathbf{e}$ in environment ρ, for a supercombinator of arity $\mathbf{d}$, and then proceeds to unwind the resulting stack.

$\mathcal{R}[\![\mathbf{e}]\!]\ \rho\ \mathbf{d} \quad = \quad \mathcal{C}[\![\mathbf{e}]\!]\ \rho \mathbin{+\!\!+} [\texttt{Update}\ \mathbf{d}, \texttt{Pop}\ \mathbf{d}, \texttt{Unwind}]$

Figure 3.6: The $\mathcal{R}$ compilation scheme for Mark 2 G-machine

We must also define a transition for `Unwind` when the top of stack item is an indirection. The effect is to replace the current stack item with the item that the indirection points to.

$$(3.17)\qquad \begin{array}{|lllll|} \hline & [\texttt{Unwind}] & \mathbf{a}_0 : \mathbf{s} & \mathbf{h}[\mathbf{a}_0 : \texttt{NInd}\ \mathbf{a}] & \mathbf{m} \\ \Longrightarrow & [\texttt{Unwind}] & \mathbf{a} : \mathbf{s} & \mathbf{h} & \mathbf{m} \\ \hline \end{array}$$

Exercise 3.9. Modify the `dispatch` function of the Mark 1 machine to incorporate the new instructions; implement the new transition rules.

3.4.3 The compiler

The only change to the compiler lies in the code generated by the $\mathcal{R}$ scheme. The new definition is given in Figure 3.6.

Exercise 3.10. Modify `compileR` to implement the new $\mathcal{R}$ scheme.

Exercise 3.11. Run the lazy evaluator on the program:

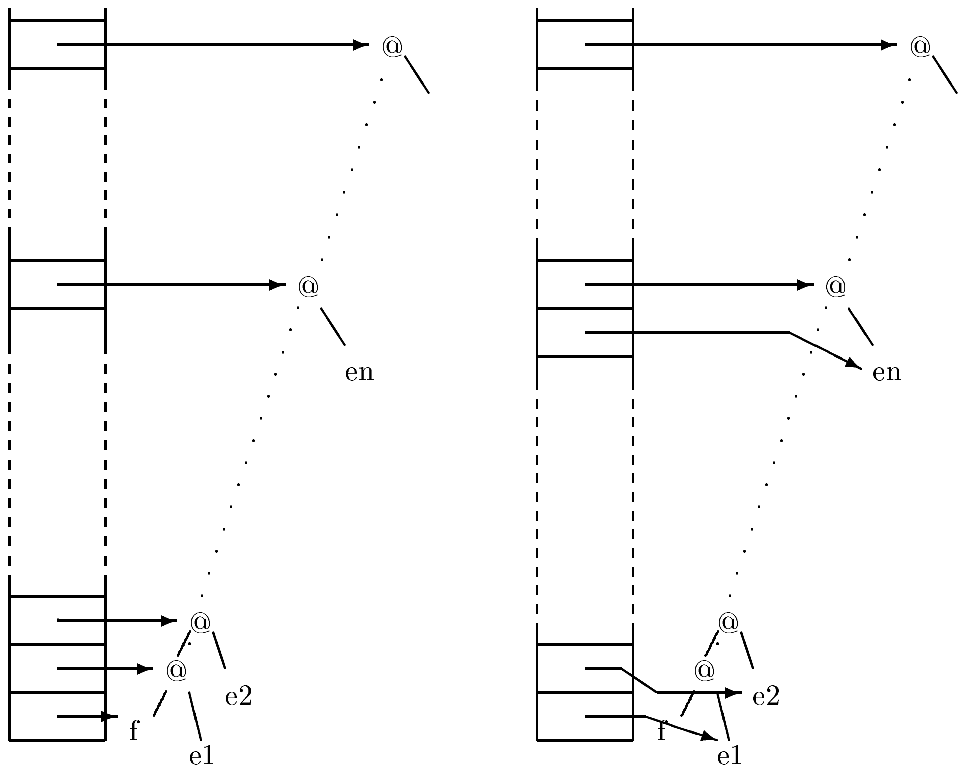
```
twice f x = f (f x)
id x = x
main = twice twice id 3
```

How many steps does it take? Why is it different from that obtained for the Mark 1 machine? Is it fair to compare the number of steps taken for the machines?

3.5 Mark 3: `let(rec)` expressions

We now extend the language so that the compiler will accept supercombinators whose body includes *`let(rec)`-bound variables*. These are represented in the data type `coreExpr` by the constructor `ELet`. It takes three arguments: a boolean flag which says whether the definitions are to be treated recursively, the definitions themselves and an expression in which the definitions are to be used.

Before we attempt to extend the machine by adding local definitions, we will have another look at the stack. In particular we will try to define a more efficient *access method* for variables. Besides the efficiency argument, we also wish to make access to locally bound variables the same as that used to bind function parameters.



Marks 1 and 2 Mark 3 onwards

Figure 3.7: Stack layout on entry to function `f`

Argument access from the stack

Suppose that the unwinding process has reached a supercombinator node `f`, and that the supercombinator takes **n** arguments. In the Mark 1 machine the stack will be in the state shown in the left-hand diagram of Figure 3.7.

Having reached the supercombinator `f`, in the Mark 3 G-machine, the stack is slightly modified. The equivalent Mark 3 stack is shown in the right-hand diagram of Figure 3.7; the top **n** elements now point directly to the expressions `e1` ... `en`. The important point here is that we have faster access to the variables (provided that the variable is accessed at least once). This is because we only look at the application node once, to get its right-hand argument.

This improves the efficiency of access to the expressions that will be substituted for the formal parameters in the supercombinator. In terms of the Mark 1 machine:

- we no longer need the function `getArg` in the `Push` instruction,
- but we do need to rearrange the stack when we `Unwind` a supercombinator with sufficient arguments.

Notice that we have retained a pointer to the root of the redex so that we can perform an `Update`.

The effects on instructions

When we choose to use the new stack layout, we necessarily have to modify certain of the machine instructions to cope. The instructions affected are `Push` and `Unwind`. The `Push` instruction will have to change because we do not need to 'look through' the application node to get at the argument.

$$
(3.18) \quad \begin{array}{|lrrrr|}
\hline
 & \texttt{Push}\ \mathbf{n} : \mathbf{i} & \mathbf{a_0} : \ldots : \mathbf{a_n} : \mathtt{s} & \mathbf{h} & \mathbf{m} \\
\Longrightarrow & \mathbf{i} & \mathbf{a_n} : \mathbf{a_0} : \ldots : \mathbf{a_n} : \mathtt{s} & \mathbf{h} & \mathbf{m} \\
\hline
\end{array}
$$

The other modification required for the new stack layout is that `Unwind` must rearrange the stack. This *rearrangement* is required whenever a supercombinator with sufficient arguments is found on the top of the stack. The new transition rule for `Unwind` is:

$$
(3.19) \quad \begin{array}{|lrrll|}
\hline
 & [\texttt{Unwind}] & \mathbf{a_0} : \ldots : \mathbf{a_n} : \mathtt{s} & \mathbf{h} \left[\begin{array}{lcl} \mathbf{a_0} & : & \texttt{NGlobal}\ \mathbf{n}\ \mathbf{c} \\ \mathbf{a_1} & : & \texttt{NAp}\ \mathbf{a_0}\ \mathbf{a'_1} \\ & \cdots & \\ \mathbf{a_n} & : & \texttt{NAp}\ \mathbf{a_{n-1}}\ \mathbf{a'_n} \end{array} \right] & \mathbf{m} \\
\Longrightarrow & \mathbf{c} & \mathbf{a'_1} : \ldots : \mathbf{a'_n} : \mathbf{a_n} : \mathtt{s} & \mathbf{h} & \mathbf{m} \\
\hline
\end{array}
$$

Notice that this definition of `Unwind` will work properly for the case where $\mathbf{n}$ is zero.

Exercise 3.12. Rewrite the `dispatch` function and the new transitions for the new instruction set. You should make use of the function `rearrange` to rearrange the stack.

```
> rearrange :: num -> gmHeap -> gmStack -> gmStack
> rearrange n heap as
>       = take n as' ++ drop n as
>       where as' = map (getArg . hLookup heap) (tl as)
```

Exercise 3.13. Test the compiler and new abstract machine on some sample programs from Appendix B, to ensure that the implementation still works.

3.5.1 Locally bound variables

Now we return to the implementation of `let(rec)` expressions, considering the non-recursive case first. The variables $\mathbf{x_1} \ldots \mathbf{x_n}$, in the expression `let` $\mathbf{x_1}$`=`$\mathbf{e_1}$`; ... ;` $\mathbf{x_n} = \mathbf{e_n}$ `in` $\mathbf{e}$, can be treated in the same way as the arguments to a supercombinator, once the expressions $\mathbf{e_1} \ldots \mathbf{e_n}$ have been created. That is, we access the variables $\mathbf{x_1} \ldots \mathbf{x_n}$ via offsets into the stack, using the environment to record their locations.

Suppose that the code to build the local definitions is called `Code`, then the sequence of actions shown in Figure 3.8 will be necessary. Initially, the stack will contain pointers to the arguments to the supercombinator. After the code to build the local definitions has executed we will have $\mathbf{n}$ new pointers on the stack. We can now proceed to build the body of the `let` expression, in a new environment that maps $\mathbf{x_i}$ to the pointer to $\mathbf{e_i}$. Finally, we need to throw away the pointers to the expressions $\mathbf{e_1} \ldots \mathbf{e_n}$ from the stack.

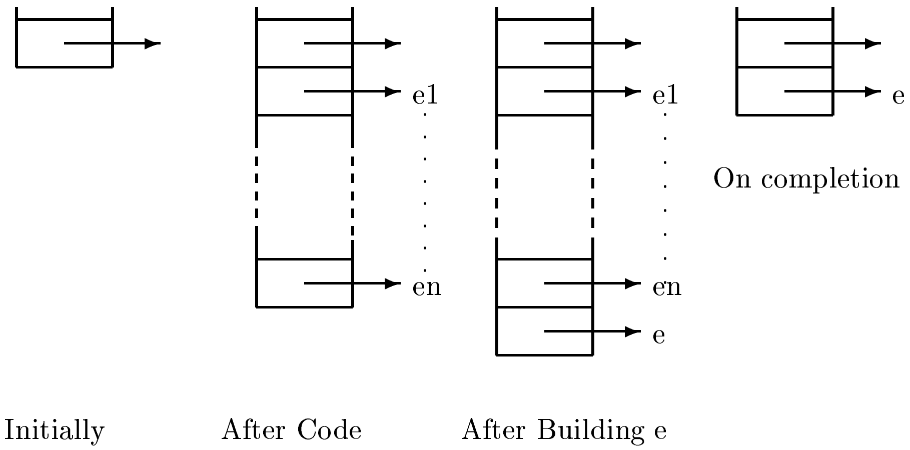


Figure 3.8: Stack usage in non-recursive local definitions

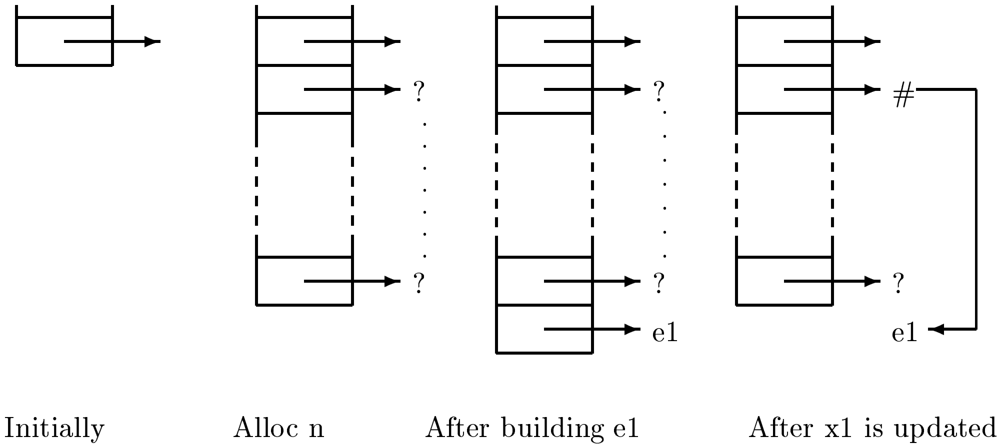


Figure 3.9: Constructing a recursively defined expression: `e1`

Because we have added $\mathbf{n}$ new variables to the stack $(\mathbf{x}_1 \ldots \mathbf{x_n})$ we must note this fact in the variable map we use to compile $\mathbf{e}$. The code to construct the local bindings – which we have called `Code` – will simply build the graph of each expression $\mathbf{e}_1 \ldots \mathbf{e_n}$ in turn, leaving the address of the piece of graph on the stack.

After building the body expression $\mathbf{e}$ – which may use any of the variables $\mathbf{x}_1 \ldots \mathbf{x_n}$ – we must remove the pointers to $\mathbf{e}_1 \ldots \mathbf{e_n}$ from the stack. This is accomplished by using a `Slide` instruction. The complete scheme for compiling a non-recursive local definition is given in Figure 3.10 (p.111).

The situation with recursive local definitions is more complicated: each of the expressions $\mathbf{e}_1 \ldots \mathbf{e_n}$ must be compiled so that the variables $\mathbf{x}_1 \ldots \mathbf{x_n}$ are in scope. To do this we create empty nodes in the graph, leaving pointers to them on the stack. Each expression $\mathbf{e}_1 \ldots \mathbf{e_n}$ is then compiled using the same variable map that we used for the compilation of the body of the non-recursive case. At the end of each expression's compiled code we place an `Update` instruction that will overwrite the empty node with the correct piece of graph. To do this we need one new instruction – `Alloc n` – which will create `n` empty graph nodes for us. In Figure 3.9 the empty graph nodes are represented by a `?` symbol.

The process shown in Figure 3.9 needs to be repeated until each of the expressions $\mathbf{e}_1 \ldots \mathbf{e}_n$ has been processed. Compiling code for the body $\mathbf{e}$ is then the same as the previous case for non-recursive local definitions. We now add the new data types for the Mark 3 machine.

3.5.2 Data structures

The instruction data type includes all of the instructions of the Mark 2 machine, with the new `Alloc` instruction and the `Slide` instruction from the Mark 1 machine.

Exercise 3.14. Modify the data type `instruction` so that it includes `Alloc` and `Slide`. You will also need to modify the function `showInstruction`, to accommodate these new instructions.

3.5.3 The evaluator

For the Mark 3 G-machine we will need to add the `Alloc` instruction which creates `n` locations in the heap. We use these locations to mark the places we will store the locally bound expressions. These nodes are initially created as indirection nodes that point to an illegal heap address: `hNull`. Because these nodes created by `Alloc` are going to be overwritten, it does not really matter what value we assign them.

$$\begin{array}{|lrrrcr|} \texttt{Alloc}\ \mathbf{n} : \mathbf{i} & & \mathbf{s} & \mathbf{h} & & \mathbf{m} \\ \Longrightarrow & \mathbf{i} & \mathbf{a}_1 : \ldots : \mathbf{a}_n : \mathbf{s} & \mathbf{h} & \left[\begin{array}{lcl} \mathbf{a}_1 & : & \texttt{NInd hNull} \\ & \cdots & \\ \mathbf{a}_n & : & \texttt{NInd hNull} \end{array}\right] & \mathbf{m} \end{array} \qquad (3.20)$$

To implement `alloc`, the transition function for the `Alloc` instruction, we use an auxiliary function `allocNodes`. Given the number of nodes required and the current heap, it returns a pair consisting of the modified heap and the list of addresses of the indirection nodes.

```
> allocNodes :: num -> gmHeap -> (gmHeap, [addr])
> allocNodes 0     heap = (heap,  [])
> allocNodes (n+1) heap = (heap2, a:as)
>                         where (heap1, as) = allocNodes n heap
>                               (heap2, a)  = hAlloc heap1 (NInd hNull)
```

Exercise 3.15. Extend the `dispatch` function, with cases for the new instructions. You should use `allocNodes` to implement `alloc`, the transition function for the `Alloc` instruction.

3.5.4 The compiler

The only change to the compiler is that there are now two more cases for which the $\mathcal{C}$ scheme can compile code. The modification to `compileC` is simple. It can now cope with a wider range of `coreExprs`. We need two new functions: `compileLetrec` and `compileLet`.

$\mathcal{C}[\![\mathbf{e}]\!]\ \rho$ generates code which constructs the graph of $\mathbf{e}$ in environment ρ, leaving a pointer to it on top of the stack.

$\mathcal{C}[\![\mathbf{f}]\!]\ \rho$	$=$	[Pushglobal $\mathbf{f}$]	where $\mathbf{f}$ is a supercombinator
$\mathcal{C}[\![\mathbf{x}]\!]\ \rho$	$=$	[Push $(\rho\ \mathbf{x})$]	where $\mathbf{x}$ is a local variable
$\mathcal{C}[\![\mathbf{i}]\!]\ \rho$	$=$	[Pushint $\mathbf{i}$]	
$\mathcal{C}[\![\mathbf{e_0}\ \mathbf{e_1}]\!]\ \rho$	$=$	$\mathcal{C}[\![\mathbf{e_1}]\!]\ \rho \mathbin{+\!\!+} \mathcal{C}[\![\mathbf{e_0}]\!]\ \rho^{+1} \mathbin{+\!\!+}$ [Mkap]	where $\rho^{+\mathbf{n}}\ \mathbf{x}$ is $(\rho\ \mathbf{x}) + \mathbf{n}$
$\mathcal{C}[\![$let $\mathbf{x_1}$=$\mathbf{e_1}$; ... ; $\mathbf{x_n}$=$\mathbf{e_n}$ in $\mathbf{e}]\!]\ \rho$			
	$=$	$\mathcal{C}[\![\mathbf{e_1}]\!]\ \rho^{+0} \mathbin{+\!\!+} \ldots \mathbin{+\!\!+}$	
		$\mathcal{C}[\![\mathbf{e_n}]\!]\ \rho^{+(\mathbf{n}-1)} \mathbin{+\!\!+}$	
		$\mathcal{C}[\![\mathbf{e}]\!]\ \rho' \mathbin{+\!\!+}$ [Slide $\mathbf{n}$]	where $\rho' = \rho^{+\mathbf{n}}[\mathbf{x_1} \mapsto \mathbf{n}-1, \ldots, \mathbf{x_n} \mapsto 0]$
$\mathcal{C}[\![$letrec $\mathbf{x_1}$=$\mathbf{e_1}$; ... ; $\mathbf{x_n}$=$\mathbf{e_n}$ in $\mathbf{e}]\!]\ \rho$			
	$=$	[Alloc $\mathbf{n}$] $\mathbin{+\!\!+}$	
		$\mathcal{C}[\![\mathbf{e_1}]\!]\ \rho' \mathbin{+\!\!+}$ [Update $\mathbf{n}-1$] $\mathbin{+\!\!+} \ldots \mathbin{+\!\!+}$	
		$\mathcal{C}[\![\mathbf{e_n}]\!]\ \rho' \mathbin{+\!\!+}$ [Update 0] $\mathbin{+\!\!+}$	
		$\mathcal{C}[\![\mathbf{e}]\!]\ \rho' \mathbin{+\!\!+}$ [Slide $\mathbf{n}$]	where $\rho' = \rho^{+\mathbf{n}}[\mathbf{x_1} \mapsto \mathbf{n}-1, \ldots, \mathbf{x_n} \mapsto 0]$

Figure 3.10: The modified $\mathcal{C}$ compilation scheme for `let` and `letrec`

```
> compileC :: gmCompiler
> compileC (EVar v)    env = [Push n],       member (aDomain env) v
>                          = [Pushglobal v], otherwise
>                            where n = aLookup env v n
> compileC (ENum n)    env = [Pushint n]
> compileC (EAp e1 e2) env = compileC e2 env ++
>                            compileC e1 (argOffset 1 env) ++
>                            [Mkap]
> compileC (ELet recursive defs e)
>                      env = compileLetrec compileC defs e env, recursive
>                          = compileLet    compileC defs e env, otherwise
```

The definition of `compileLet` follows the specification given in Figure 3.10. It takes as arguments: the compilation scheme `comp` for the body `e`, the definitions `defs` and the current environment `env`. We have provided the compiler parameter so that in later versions of the machine we do not have to rewrite this function.

```
> compileLet :: gmCompiler -> [(name, coreExpr)] -> gmCompiler
> compileLet comp defs expr env
>    = compileLet' defs env ++ comp expr env' ++ [Slide (#defs)]
>      where env' = compileArgs defs env
```

The compilation of the new definitions is accomplished by the function `compileLet'`.

```
> compileLet' :: [(name, coreExpr)] -> gmEnvironment -> gmCode
> compileLet' []                  env = []
> compileLet' ((name, expr):defs) env
```

```
>     = compileC expr env ++ compileLet' defs (argOffset 1 env)
```

compileLet also uses compileArgs to modify the offsets into the stack for the compilation of the body, e.

```
> compileArgs :: [(name, coreExpr)] -> gmEnvironment -> gmEnvironment
> compileArgs defs env
>      = zip2 (map first defs) [n-1, n-2 .. 0] ++ argOffset n env
>      where n = #defs
```

An example

In this example we will show how the code for the fixpoint combinator Y is compiled. The definition we will use is:

```
Y f = letrec x = f x in x
```

This is the so-called 'knot-tying' fixpoint combinator; we will see why it has this name when we run the resulting code. When the above definition is compiled, the compileSc function will need to produce code for the supercombinator.

```
compileSc ("Y", ["f"], ELet True [("x", EAp (EVar "f") (EVar "x"))] (EVar "x"))
```

This in turn calls the compileR function with an environment for the variable f; having first created a name for the supercombinator (Y) and its number of arguments (1).

```
("Y", 1, compileR e [("f", 0)])
where e = ELet True [("x", EAp (EVar "f") (EVar "x"))] (EVar "x")
```

For convenience we will refer to the body of the expression as e. The function compileR calls compileC, placing the tidying-up code at the end.

```
("Y", 1, compileC e [("f", 0)] ++ [Update 1, Pop 1, Unwind])
```

Referring to the compilation scheme in Figure 3.10, we see that to compile a letrec we first create a new environment. In the figure this is called ρ'; in this example we will call it p. It is an extension of the initial environment in which we also give a stack location to the local variable x.

```
("Y", 1, [Alloc 1] ++
         compileC (EAp (EVar "f") (EVar "x")) p ++ [Update 0] ++
         compileC (EVar "x") p ++ [Slide 1] ++
         [Update 1, Pop 1, Unwind])
         where p = [("x", 0), ("f", 1)]
```

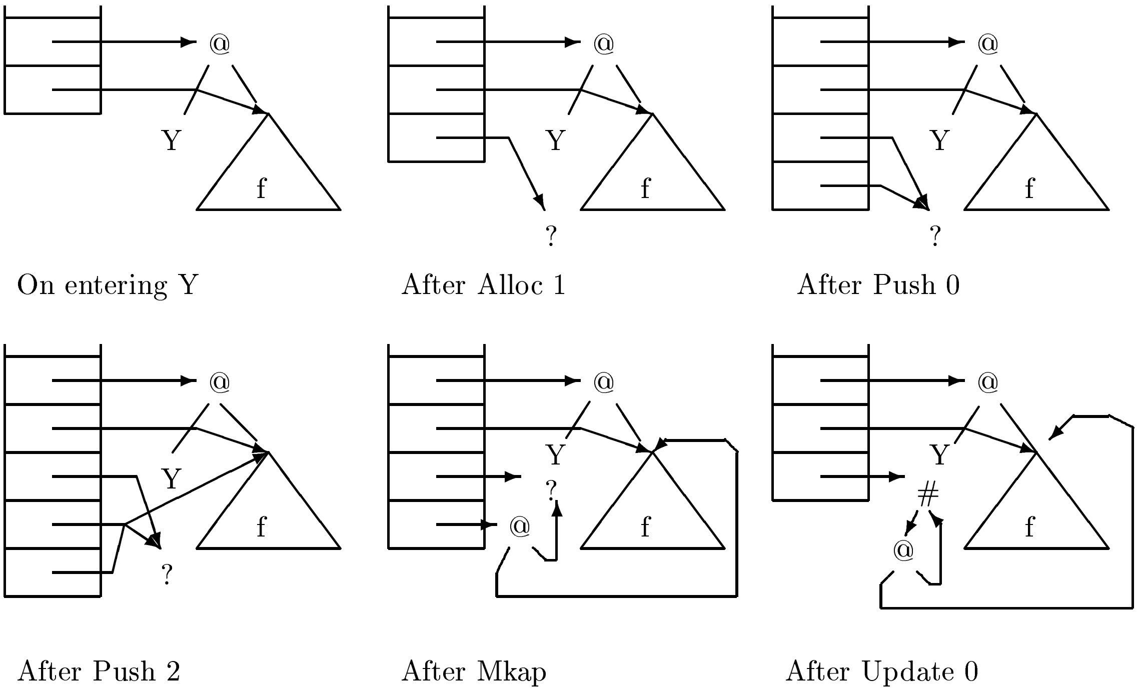


Figure 3.11: Execution of code for `Y`

The code generation is laid out in the same fashion as the compilation scheme. When the expressions involving `compileC` are simplified we get:

```
("Y", 1, [Alloc 1] ++
         [Push 0, Push 2, Mkap] ++ [Update 0] ++
         [Push 0] ++ [Slide 1] ++
         [Update 1, Pop 1, Unwind])
```

Which gives the following code sequence:

```
("Y", 1, [Alloc 1, Push 0, Push 2, Mkap, Update 0, Push 0,
          Slide 1, Update 1, Pop 1, Unwind])
```

We can see the way in which this code executes in Figure 3.11. This definition of the `Y` supercombinator is called 'knot-tying' because we are tying a knot in the graph when we do the `Update 0` as the fifth instruction. We have not shown the remainder of the instructions, as this is left as Exercise 3.18.

Exercise 3.16. The compilation of `letrec`s is defined in Figure 3.10. Implement the function `compileLetrec` to perform this operation.

Exercise 3.17. What test programs would you use to show that the new compiler and instruction set work properly?

Exercise 3.18. By running the code generated for the supercombinator `Y`, or otherwise, draw the remainder of the state transitions in the style of Figure 3.11.

Exercise 3.19. Give a shorter, alternative, code sequence for the supercombinator `Y`. It should still construct a 'knot-tying' version.

Exercise 3.20. In the absence of a `letrec` construct in the language, how would you define the fixpoint combinator `Y`? How is this definition different from the one we used in the example?

3.6 Mark 4: Adding primitives

In this section we add primitive operations to the G-machine; this makes it useful. By primitive operations we mean operations like addition, multiplication and so on. We will use addition as a running example throughout this section.

The addition instruction will be called `Add`; which adds two numbers from the heap, placing the result into a new node in the heap. The addresses of the two arguments are on top of the stack, and this is where the address of the result is subsequently placed. It has the following transition rule.

(3.21)

	$\texttt{Add} : \mathbf{i}$	$\mathbf{a}_0 : \mathbf{a}_1 : \mathbf{s}$	$\mathbf{h}[\mathbf{a}_0 : \texttt{NNum}\ \mathbf{n}_0,\ \mathbf{a}_1 : \texttt{NNum}\ \mathbf{n}_1]$	$\mathbf{m}$
$\Longrightarrow$	$\mathbf{i}$	$\mathbf{a} : \mathbf{s}$	$\mathbf{h}[\mathbf{a} : \texttt{NNum}\ \mathbf{n}_0 + \mathbf{n}_1]$	$\mathbf{m}$

We could continue to expand the G-machine with other instructions to implement the remainder of the operations required, but before we do, let us pause to consider whether we have missed something here. The problem is that the rule only applies if the two objects on top of the stack are numbers. Since we are working on a machine that supports lazy evaluation there is no good reason to suppose that this will always be the case. In the template machine `Add` checked that its arguments were evaluated. In the G-machine we want to keep the instructions simple, so we will only use `Add` in situations where we guarantee that the arguments are already evaluated.

What we do instead is to augment the instruction set further with an `Eval` instruction. This satisfies the following constraint:

Suppose that we are in a state:

$$\boxed{\textit{Eval} : \mathbf{i} \quad \mathbf{a} : \mathbf{s} \quad \mathbf{h} \quad \mathbf{m}}$$

Whenever execution resumes with instruction sequence **i**, *the state will be:*

$$\boxed{\mathbf{i} \quad \mathbf{a} : \mathbf{s} \quad \mathbf{h}' \quad \mathbf{m}}$$

and the item on top of the stack will be in WHNF.

It is also possible for ***Eval*** *to fail to terminate; this will be the case when the node pointed to from the top of the stack has no WHNF.*

If the node whose address is on top of the stack is already in WHNF, then the `Eval` instruction does nothing. If there is reduction to be performed, then the action of `Eval` is to perform an evaluation to WHNF. If this call terminates then execution resumes with nothing except the heap component changed. This is similar to the structure of *subroutine call and return* traditionally used in programming language implementation. We recall that the classic way to implement this feature is to use a stack. The stack will save sufficient of the machine's current context that it can resume when the subroutine call completes.

In the Mark 4 G-machine this stack is called the *dump*, and is a stack of pairs, whose first component is a code sequence, and whose second component is a stack. This is similar to the dump in the template machine (see Section 2.6), except we now have to restore the original code sequence as well as the original stack. Hence both components are kept on the dump.

3.6.1 Data structures

We extend the G-machine state by adding a dump component. As previously discussed, this is used to implement recursive calls to the evaluator.

```
> gmState == (gmCode,              || Current instruction stream
>             gmStack,             || Current stack
>             gmDump,              || Current dump
>             gmHeap,              || Heap of nodes
>             gmGlobals,           || Global addresses in heap
>             gmStats)             || Statistics
```

The dump itself is a stack of `dumpItem`. Each of these is a pair consisting of the instruction stream and stack to use when we resume the original computation.

```
> gmDump     == [gmDumpItem]
> gmDumpItem == (gmCode, gmStack)
```

When we add this new component we must change all of the previously specified access functions. We must also add access functions for the dump.

```
> getDump :: gmState -> gmDump
> getDump (i, stack, dump, heap, globals, stats) = dump


> putDump :: gmDump -> gmState -> gmState
> putDump dump' (i, stack, dump, heap, globals, stats)
>    = (i, stack, dump', heap, globals, stats)
```

Notice that it is only in the access functions that we have done pattern matching on G-machine states. Changes to other functions as a result of adding new components to the state are no longer needed.

Exercise 3.21. Make the relevant changes to the other access functions.

In addition to the new definition of state, we also need some new instructions. We reuse all of the instructions from the Mark 3 machine.

```
> instruction ::= Slide num
>            | Alloc num
>            | Update num
>            | Pop num
>            | Unwind
>            | Pushglobal name
>            | Pushint num
>            | Push num
>            | Mkap
```

In addition we include the Eval instruction,

```
>             | Eval
```

the following arithmetic instructions:

```
>             | Add | Sub | Mul | Div | Neg
```

and the following comparison instructions:

```
>             | Eq | Ne | Lt | Le | Gt | Ge
```

We also include a primitive form of conditional in the Cond instruction.

```
>            | Cond gmCode gmCode
```

Exercise 3.22. Add cases to showInstruction to print all of the new instructions.

3.6.2 Printing the state

We take this opportunity to revise the definition of showState, so that it displays the dump component.

```
> showState :: gmState -> iseq
> showState s
>    = iConcat       [showStack s,                   iNewline,
>                     showDump s,                    iNewline,
>                     showInstructions (getCode s), iNewline]
```

We therefore need to define showDump.

```
> showDump :: gmState -> iseq
> showDump s
>     = iConcat       [iStr "  Dump:[",
>                      iIndent (iInterleave iNewline
>                             (map showDumpItem (reverse (getDump s)))),
>                      iStr "]"]
```

This in turn needs the function `showDumpItem`.

```
> showDumpItem :: gmDumpItem -> iseq
> showDumpItem (code, stack)
>     = iConcat       [iStr "<",
>                      shortShowInstructions 3 code, iStr ", ",
>                      shortShowStack stack,         iStr ">"]
```

We use the function `shortShowInstructions` to print only the first three instructions of the instruction stream in the dump items. This is usually sufficient to indicate where the computation will resume.

```
> shortShowInstructions :: num -> gmCode -> iseq
> shortShowInstructions number code
>     = iConcat [iStr "{", iInterleave (iStr "; ") dotcodes, iStr "}"]
>     where   codes   = map showInstruction (take number code)
>             dotcodes = codes ++ [iStr "..."], #code > number
>                      = codes,                 otherwise
```

Similarly, we do not need the full details of the stack component of the dump item either, so we use `shortShowStack`.

```
> shortShowStack :: gmStack -> iseq
> shortShowStack stack
>     = iConcat [iStr "[",
>             iInterleave (iStr ", ") (map (iStr . showaddr) stack),
>             iStr "]"]
```

3.6.3 The new instruction transitions

Evaluator instructions

There are actually very few instructions that manipulate the dump. First, there is `Eval` itself, which creates a new dump item whenever the node on top of the stack is not in WHNF. Secondly, there is a modification to the `Unwind` instruction that pops a dump item when an evaluation is completed.

We first describe the new `Unwind` instruction. When the expression held in the stack is in WHNF, `Unwind` can restore the old context from the dump, placing the last address in the stack on the restored old stack. We see this clearly in the transition for the case of numbers.[1]

$$(3.22)\quad \boxed{\begin{array}{llllll} & [\texttt{Unwind}] & \mathbf{a}:\mathbf{s} & \langle \mathbf{i}', \mathbf{s}'\rangle : \mathbf{d} & \mathbf{h}[\mathbf{a}:\texttt{NNum}\ \mathbf{n}] & \mathbf{m} \\ \Longrightarrow & \mathbf{i}' & \mathbf{a}:\mathbf{s}' & \mathbf{d} & \mathbf{h} & \mathbf{m} \end{array}}$$

[1]The rule only applies if the dump is non-empty; if the dump is empty then the machine has terminated.

The expression with address $\mathbf{a}$ is in WHNF because it is an integer, so we restore the old instruction sequence $\mathbf{i}'$ and the stack is now the old stack $\mathbf{s}'$ with the address $\mathbf{a}$ on top. All other transitions for `Unwind` remain the same as they were in the Mark 3 machine (except that they have the dump component in their state).

We are now in a position to specify the rule for `Eval`. It saves the remainder of the stack $\mathbf{s}$ and the rest of the instructions $\mathbf{i}$ as a dump item on the dump. The new code sequence is just unwinding and the new stack contains the singleton $\mathbf{a}$.

$$(3.23)\qquad \boxed{\begin{array}{lrrrrr} & \texttt{Eval}:\mathbf{i} & \mathbf{a}:\mathbf{s} & \mathbf{d} & \mathbf{h} & \mathbf{m} \\ \Longrightarrow & [\texttt{Unwind}] & [\mathbf{a}] & \langle \mathbf{i}, \mathbf{s}\rangle : \mathbf{d} & \mathbf{h} & \mathbf{m} \end{array}}$$

Arithmetic instructions

The *dyadic arithmetic operators* all have the following generic transition rule. Let us suppose that the arithmetic operator we wish to implement is $\odot$; the transition rule for the instruction $\odot$ is then:

$$(3.24)\qquad \boxed{\begin{array}{lrrrlr} & \odot:\mathbf{i} & \mathbf{a}_0:\mathbf{a}_1:\mathbf{s} & \mathbf{d} & \mathbf{h}[\mathbf{a}_0:\texttt{NNum}\ \mathbf{n}_0,\ \mathbf{a}_1:\texttt{NNum}\ \mathbf{n}_1] & \mathbf{m} \\ \Longrightarrow & \mathbf{i} & \mathbf{a}:\mathbf{s} & \mathbf{d} & \mathbf{h}[\mathbf{a}:\texttt{NNum}\ (\mathbf{n}_0 \odot \mathbf{n}_1)] & \mathbf{m} \end{array}}$$

What has happened is that the two numbers on top of the stack have had the dyadic operator $\odot$ applied to them. The result, which is entered into the heap, has its address placed on the stack. The `Neg` instruction negates the number on top of the stack, so it has transition rule:

$$(3.25)\qquad \boxed{\begin{array}{lrrrlr} & \texttt{Neg}:\mathbf{i} & \mathbf{a}:\mathbf{s} & \mathbf{d} & \mathbf{h}[\mathbf{a}:\texttt{NNum}\ \mathbf{n}] & \mathbf{m} \\ \Longrightarrow & \mathbf{i} & \mathbf{a}':\mathbf{s} & \mathbf{d} & \mathbf{h}[\mathbf{a}':\texttt{NNum}\ (-\mathbf{n})] & \mathbf{m} \end{array}}$$

Notice how similar all of the dyadic operations are. First we extract the two numbers from the heap, then we perform the operation, and finally we place the answer back in the heap. This suggests that we should write some higher-order functions that are parameterised over the extraction from heap (which we call 'unboxing' the value), and insertion back into the heap (which we call 'boxing' the value), along with the specific operation we wish to perform.

Let us write the boxing operations first. `boxInteger` takes a number and an initial state, and returns a new state in which the number has been placed into the heap, and a pointer to this new node left on top of the stack.

```
> boxInteger :: num -> gmState -> gmState
> boxInteger n state
>       = putStack (a: getStack state) (putHeap h' state)
>       where (h', a) = hAlloc (getHeap state) (NNum n)
```

Now to extract an integer at address `a` from a state, we will use `unboxInteger`.

```
> unboxInteger :: addr -> gmState -> num
```

```
> unboxInteger a state
>       = ub (hLookup (getHeap state) a)
>       where   ub (NNum i) = i
>               ub n        = error "Unboxing a non-integer"
```

A generic *monadic operator* can now be specified in terms of its boxing function, `box`, its unboxing function `unbox`, and the operator `op` on the unboxed values.

```
> primitive1 :: (** -> gmState -> gmState) || boxing function
>               -> (addr -> gmState -> *) || unboxing function
>               -> (* ->  **)             || operator
>               -> (gmState -> gmState)   || state transition
```

```
> primitive1 box unbox op state
>    = box (op (unbox a state)) (putStack as state)
>    where (a:as) = getStack state
```

The generic *dyadic operators* can now be implemented in a similar way using `primitive2`.

```
> primitive2 :: (** -> gmState -> gmState) || boxing function
>               -> (addr -> gmState -> *) || unboxing function
>               -> (* ->  * -> **)        || operator
>               -> (gmState -> gmState)   || state transition
```

```
> primitive2 box unbox op state
>    = box (op (unbox a0 state) (unbox a1 state)) (putStack as state)
>    where (a0:a1:as) = getStack state
```

To be even more explicit, `arithmetic1` implements all *monadic arithmetic*, and `arithmetic2` implements all *dyadic arithmetic*.

```
> arithmetic1 :: (num -> num)             || arithmetic operator
>               -> (gmState -> gmState)   || state transition
```

```
> arithmetic1 = primitive1 boxInteger unboxInteger
```

```
> arithmetic2 :: (num -> num -> num)      || arithmetic operator
>               -> (gmState -> gmState)   || state transition
```

```
> arithmetic2 = primitive2 boxInteger unboxInteger
```

As the alert reader would expect, we will be taking advantage of the generality of these functions later in the chapter.

Exercise 3.23. Implement all of the new instruction transitions for the machine. Modify the `dispatch` function to deal with the new instructions. You should use the higher-order functions `primitive1` and `primitive2` to implement the operators.

Exercise 3.24. Why are indirection nodes never left on top of the stack on completing an `Eval` instruction?

Comparison instructions

The *comparison operators* all have the following generic transition rule. Let us suppose that the comparison operator we wish to implement is $\odot$; the transition rule for the instruction $\odot$ is then:

$$(3.26)\quad \boxed{\begin{array}{lrrll}
 & \odot : \mathbf{i} & \mathbf{a}_0 : \mathbf{a}_1 : \mathbf{s} & \mathbf{d} & \mathbf{h}[\mathbf{a}_0 : \texttt{NNum}\ \mathbf{n}_0,\ \mathbf{a}_1 : \texttt{NNum}\ \mathbf{n}_1] \quad \mathbf{m} \\
\Longrightarrow & \mathbf{i} & \mathbf{a} : \mathbf{s} & \mathbf{d} & \mathbf{h}[\mathbf{a} : \texttt{NNum}\ (\mathbf{n}_0 \odot \mathbf{n}_1)] \qquad\quad \mathbf{m}
\end{array}}$$

What has happened is that the two numbers on top of the stack have had the dyadic operator $\odot$ applied to them. The result, which is entered into the heap, has its address placed on the stack. This is almost the same as arithmetic.

The difference is that an operation, `==` say, returns a boolean and not an integer. To fix this we turn booleans into integers using the following rule:

- we represent `True` by the integer `1`;
- we represent `False` by the integer `0`.

To make the use of `primitive2` possible, we define `boxBoolean`

```
> boxBoolean :: bool -> gmState -> gmState
> boxBoolean b state
>    = putStack (a: getStack state) (putHeap h' state)
>    where   (h',a) = hAlloc (getHeap state) (NNum b')
>            b'     = 1, b
>                   = 0, otherwise
```

Using this definition we can write a generic comparison function, which we call `comparison`. This function takes a *boxing* function for the booleans, the unboxing function for integers (`unboxInteger`), and a comparison operator; it returns a state transition.

```
> comparison :: (num -> num -> bool) -> gmState -> gmState
> comparison = primitive2 boxBoolean unboxInteger
```

Finally, we implement the `Cond` instruction, which we will use to compile the `if` function. It has two transition rules:

$$(3.27)\quad \boxed{\begin{array}{lrrrlr} & \texttt{Cond}\ \mathbf{i}_1\ \mathbf{i}_2 : \mathbf{i} & \mathbf{a} : \mathbf{s} & \mathbf{d} & \mathbf{h}[\mathbf{a} : \texttt{NNum}\ 1] & \mathbf{m} \\ \Longrightarrow & \mathbf{i}_1 \mathbin{+\!\!+} \mathbf{i} & \mathbf{s} & \mathbf{d} & \mathbf{h} & \mathbf{m} \end{array}}$$

In the first case – where there is the number `1` on top of the stack – we take the first branch. This means that we execute the instructions $\mathbf{i}_1$ before continuing to execute the instructions $\mathbf{i}$.

$$(3.28)\quad \boxed{\begin{array}{lrrrlr} & \texttt{Cond}\ \mathbf{i}_1\ \mathbf{i}_2 : \mathbf{i} & \mathbf{a} : \mathbf{s} & \mathbf{d} & \mathbf{h}[\mathbf{a} : \texttt{NNum}\ 0] & \mathbf{m} \\ \Longrightarrow & \mathbf{i}_2 \mathbin{+\!\!+} \mathbf{i} & \mathbf{s} & \mathbf{d} & \mathbf{h} & \mathbf{m} \end{array}}$$

Alternatively, if the number on top of the stack is `0`, we execute the instruction sequence $\mathbf{i}_2$ first, and then the sequence $\mathbf{i}$.

Exercise 3.25. Implement the transitions for the comparison instructions and the `Cond` instruction.

3.6.4 The compiler

The compiler will eventually need to be changed to take advantage of these new instructions to compile arithmetic expressions. For the moment we make only the minimum set of changes that will allow us to use the arithmetic instructions we have so laboriously added. First, the `compile` function must create a new initial state in which the initial dump is empty and in which the initial code sequence differs from the one we have used so far.

```
> compile :: coreProgram -> gmState
> compile program
>    = (initialCode, [], [], heap, globals, statInitial)
>    where (heap, globals) = buildInitialHeap program
```

```
> initialCode :: gmCode
> initialCode = [Pushglobal "main", Eval]
```

Exercise 3.26. Why has the initial instruction sequence been changed? What happens if we retain the old one?

The simplest way to extend the compiler is simply to add G-machine code for each of the new built-in functions to the `compiledPrimitives`. The initial four instructions of the sequence ensure that the arguments have been evaluated to integers.

```
> compiledPrimitives :: [gmCompiledSC]
> compiledPrimitives
>    =       [("+", 2, [Push 1, Eval, Push 1, Eval, Add, Update 2, Pop 2, Unwind]),
>             ("-", 2, [Push 1, Eval, Push 1, Eval, Sub, Update 2, Pop 2, Unwind]),
>             ("*", 2, [Push 1, Eval, Push 1, Eval, Mul, Update 2, Pop 2, Unwind]),
>             ("/", 2, [Push 1, Eval, Push 1, Eval, Div, Update 2, Pop 2, Unwind]),
```

We also need to add the negation function. As this only takes one argument, we only evaluate one argument.

```
>              ("negate", 1, [Push 0, Eval, Neg, Update 1, Pop 1, Unwind]),
```

The comparison operations are implemented as follows.

```
>              ("==", 2, [Push 1, Eval, Push 1, Eval, Eq, Update 2, Pop 2, Unwind]),
>              ("~=", 2, [Push 1, Eval, Push 1, Eval, Ne, Update 2, Pop 2, Unwind]),
>              ("<",  2, [Push 1, Eval, Push 1, Eval, Lt, Update 2, Pop 2, Unwind]),
>              ("<=", 2, [Push 1, Eval, Push 1, Eval, Le, Update 2, Pop 2, Unwind]),
>              (">",  2, [Push 1, Eval, Push 1, Eval, Gt, Update 2, Pop 2, Unwind]),
>              (">=", 2, [Push 1, Eval, Push 1, Eval, Ge, Update 2, Pop 2, Unwind]),
```

The `if` function is compiled so that it uses `Cond` for the branching.

```
>              ("if", 3, [Push 0, Eval, Cond [Push 1] [Push 2],
>                                 Update 3, Pop 3, Unwind])]
```

Exercise 3.27. What test programs from Appendix B would you use in order to check that the new instructions and compiler work?

3.7 Mark 5: Towards better handling of arithmetic

The way the G-machine is implemented at the moment, each arithmetic operator is called via one of the compiled primitives. We can improve on this arrangement by observing that often we can call the arithmetic operator directly. For example, consider the following simple program:

```
main = 3+4*5
```

This generates the following code when we use the current compiler:

```
[Pushint 5, Pushint 4, Pushglobal "*", Mkap, Mkap,
 Pushint 3, Pushglobal "+", Mkap, Mkap, Eval]
```

When executed this code will take 33 steps and use 11 heap nodes. Our first thought must surely be that we can use the instructions `Add` and `Mul` in place of calls to the functions '`+`' and '`*`'. This leads to the following improved code:

```
[Pushint 5, Pushint 4, Mul, Pushint 3, Add]
```

This will take only five steps to execute and uses five heap nodes.

3.7.1 A problem

A possible problem arises when we consider our next example program.

```
main = K 1 (1/0)
```

This generates the following code:

```
[Pushint 0, Pushint 1, Pushglobal "/", Mkap, Mkap,
 Pushint 1, Pushglobal "K", Mkap, Mkap, Eval]
```

If we follow the pattern of the previous example we might try generating the code:

```
[Pushint 0, Pushint 1, Div,
 Pushint 1, Pushglobal "K", Mkap, Mkap, Eval]
```

The problem is that the division operator is applied *before* we reduce K, with the result that a *division-by-zero error* is generated. A correct compiler must generate code that will not give such errors.

What has happened is that our code is too strict. The code is evaluating expressions that it need not – which results in errors arising where they should not. A similar problem will also arise when *non-terminating expressions* are inadvertently evaluated.

3.7.2 The solution

The solution to the problem is to keep track of the context in which an expression appears. We will distinguish two contexts[2]:

Strict The value of the expression will be required in WHNF.

Lazy The value of the expression may or may not be required in WHNF.

Corresponding to each context, we have a compilation scheme which will compile an expression to a sequence of G-machine instructions. In the strict context this compilation scheme is the $\mathcal{E}$ scheme; in the lazy context we will use the $\mathcal{C}$ scheme we have seen already.

We would like to find as many strict contexts as possible, since these contexts allow us to generate better code. We make the following observation: whenever a supercombinator is instantiated it is because we wish to evaluate its value to WHNF. From this we conclude that the body of a supercombinator can always be evaluated in a strict context. There are also expressions where we know that some sub-expressions will be evaluated to WHNF if the expression is evaluated to WHNF.

The class of strict context expressions can be described recursively.

[2]It is possible to distinguish more contexts. Projection analysis [Wadler 1987] and evaluation transformers [Burn 1991] are two ways to do this.

- The expression in the body of a supercombinator definition is in a strict context.
- If $\mathbf{e}_0 \odot \mathbf{e}_1$ occurs in a strict context, where $\odot$ is an arithmetic or comparison operator, then the expressions $\mathbf{e}_0$ and $\mathbf{e}_1$ are also in a strict context.
- If `negate` $\mathbf{e}$ occurs in a strict context, then the expression $\mathbf{e}$ also does.
- If the expression `if` $\mathbf{e}_0$ $\mathbf{e}_1$ $\mathbf{e}_2$ occurs in a strict context, then so do the expressions $\mathbf{e}_0$, $\mathbf{e}_1$, and $\mathbf{e}_2$.
- If `let(rec)` Δ `in` $\mathbf{e}$ occurs in a strict context then the expression $\mathbf{e}$ is also in a strict context.

An example should make this clear; consider the body of the supercombinator `f`:

```
f x y = (x+y) + g (x*y)
```

Both `(x+y)` and `g (x*y)` will be evaluated in a strict context – because the body of the supercombinator is. In the first case this causes `x` and `y` to be evaluated in a strict context – because `+` propagates the strict context. In the second expression, the presence of a user-defined supercombinator means that the sub-expression `x*y` will be compiled assuming that the expression may not be evaluated.

This suggests that we can implement the strict-context compiler, $\mathcal{E}$, in a recursive manner. Because the body of each supercombinator is evaluated in a strict context we will need to call the $\mathcal{E}$ scheme function from the $\mathcal{R}$ scheme. This satisfies the first of the points above. To propagate the context information into sub-expressions we will recursively invoke the $\mathcal{E}$ scheme for arithmetic expressions.

The new compiler schemes are defined in Figure 3.12.

To make it easier to extend the set of built-in operators that can be compiled using the new scheme, we define `builtInDyadic`.

```
> builtInDyadic :: assoc name instruction
> builtInDyadic
>     =       [("+", Add), ("-", Sub), ("*", Mul), ("div", Div),
>              ("==", Eq), ("~=", Ne), (">=", Ge),
>              (">",  Gt), ("<=", Le), ("<",  Lt)]
```

Exercise 3.28. Modify the existing compiler functions `compileR` and `compileE` so that they implement the $\mathcal{R}$ scheme and $\mathcal{E}$ scheme of Figure 3.12. You should use `builtInDyadic`.

Occasionally the machine will fail when the new compiler is used. What we need is to introduce a new rule for the `Unwind` instruction.

(3.29)

	[`Unwind`]	$[\mathbf{a}_0, \ldots, \mathbf{a_k}]$	$\langle \mathbf{i}, \mathbf{s} \rangle : \mathbf{d}$	$\mathbf{h}[\mathbf{a}_0 : \texttt{NGlobal}\ \mathbf{n}\ \mathbf{c}]$	$\mathbf{m}$
$\Longrightarrow$	$\mathbf{i}$	$\mathbf{a_k} : \mathbf{s}$	$\mathbf{d}$	$\mathbf{h}$	$\mathbf{m}$ when $\mathbf{k} < \mathbf{n}$

This allows us to use `Eval` to evaluate any object to WHNF, and not just numbers.

$\mathcal{R}[\![\mathbf{e}]\!]\ \rho\ \mathbf{d}$ generates code which instantiates the expression $\mathbf{e}$ in environment ρ, for a supercombinator of arity $\mathbf{d}$, and then proceeds to unwind the resulting stack.

$$\mathcal{R}[\![\mathbf{e}]\!]\ \rho\ \mathbf{d} \quad = \quad \mathcal{E}[\![\mathbf{e}]\!]\ \rho \mathbin{+\!\!+} [\texttt{Update}\ \mathbf{d}, \texttt{Pop}\ \mathbf{d}, \texttt{Unwind}]$$

$\mathcal{E}[\![\mathbf{e}]\!]\ \rho$ compiles code that evaluates an expression $\mathbf{e}$ to WHNF in environment ρ, leaving a pointer to the expression on top of the stack.

$$\begin{array}{rcl}
\mathcal{E}[\![\mathbf{i}]\!]\ \rho & = & [\texttt{Pushint}\ \mathbf{i}] \\
\multicolumn{3}{l}{\mathcal{E}[\![\texttt{let}\ \mathbf{x}_1\texttt{=}\mathbf{e}_1\texttt{;}\ \ldots\texttt{;}\ \mathbf{x}_\mathbf{n}\texttt{=}\mathbf{e}_\mathbf{n}\ \texttt{in}\ \mathbf{e}]\!]\ \rho} \\
 & = & \mathcal{C}[\![\mathbf{e}_1]\!]\ \rho^{+0} \mathbin{+\!\!+} \ldots \mathbin{+\!\!+} \\
 & & \mathcal{C}[\![\mathbf{e}_\mathbf{n}]\!]\ \rho^{+(\mathbf{n}-1)} \mathbin{+\!\!+} \\
 & & \mathcal{E}[\![\mathbf{e}]\!]\ \rho' \mathbin{+\!\!+} [\texttt{Slide}\ \mathbf{n}] \\
 & & \qquad\text{where } \rho' = \rho^{+\mathbf{n}}[\mathbf{x}_1 \mapsto \mathbf{n}-1, \ldots, \mathbf{x}_\mathbf{n} \mapsto 0] \\
\multicolumn{3}{l}{\mathcal{E}[\![\texttt{letrec}\ \mathbf{x}_1\texttt{=}\mathbf{e}_1\texttt{;}\ \ldots\texttt{;}\ \mathbf{x}_\mathbf{n}\texttt{=}\mathbf{e}_\mathbf{n}\ \texttt{in}\ \mathbf{e}]\!]\ \rho} \\
 & = & [\texttt{Alloc}\ \mathbf{n}] \mathbin{+\!\!+} \\
 & & \mathcal{C}[\![\mathbf{e}_1]\!]\ \rho' \mathbin{+\!\!+} [\texttt{Update}\ \mathbf{n}-1] \mathbin{+\!\!+} \ldots \mathbin{+\!\!+} \\
 & & \mathcal{C}[\![\mathbf{e}_\mathbf{n}]\!]\ \rho' \mathbin{+\!\!+} [\texttt{Update}\ 0] \mathbin{+\!\!+} \\
 & & \mathcal{E}[\![\mathbf{e}]\!]\ \rho' \mathbin{+\!\!+} [\texttt{Slide}\ \mathbf{n}] \\
 & & \qquad\text{where } \rho' = \rho^{+\mathbf{n}}[\mathbf{x}_1 \mapsto \mathbf{n}-1, \ldots, \mathbf{x}_\mathbf{n} \mapsto 0] \\
\mathcal{E}[\![\mathbf{e}_0\ \texttt{+}\ \mathbf{e}_1]\!]\ \rho & = & \mathcal{E}[\![\mathbf{e}_1]\!]\ \rho \mathbin{+\!\!+} \mathcal{E}[\![\mathbf{e}_0]\!]\ \rho^{+1} \mathbin{+\!\!+} [\texttt{Add}] \\
 & & \text{And similarly for other arithmetic and comparison expressions} \\
\mathcal{E}[\![\texttt{negate}\ \mathbf{e}]\!]\ \rho & = & \mathcal{E}[\![\mathbf{e}]\!]\ \rho \mathbin{+\!\!+} [\texttt{Neg}] \\
\mathcal{E}[\![\texttt{if}\ \mathbf{e}_0\ \mathbf{e}_1\ \mathbf{e}_2]\!]\ \rho & = & \mathcal{E}[\![\mathbf{e}_0]\!]\ \rho \mathbin{+\!\!+} [\texttt{Cond}\ (\mathcal{E}[\![\mathbf{e}_1]\!]\ \rho)\ (\mathcal{E}[\![\mathbf{e}_2]\!]\ \rho)] \\
\mathcal{E}[\![\mathbf{e}]\!]\ \rho & = & \mathcal{C}[\![\mathbf{e}]\!]\ \rho \mathbin{+\!\!+} [\texttt{Eval}] \qquad \text{the default case}
\end{array}$$

Figure 3.12: The $\mathcal{R}$ and $\mathcal{E}$ compilation schemes for the Mark 5 machine

Exercise 3.29. Implement the new transition for `Unwind`. Write a program that fails without the new `Unwind` transition.

Exercise 3.30. Compare the execution of the example program used at the start of this section, with its execution on the Mark 4 machine.

```
main = 3+4*5
```

Try some other programs from Appendix B.

The way we implemented the strict context in the compiler is addis simple example of the *inherited attributes* from compiler theory. If we regard the strict context as an attribute of an expression, then a sub-expression *inherits* its strict context from its parent expression. The general theory is discussed in [Aho *et al.* 1986].

It is unfortunately not possible – in general – to determine at compile-time whether an expression should be compiled with a strict context. We therefore have to accept a compromise. In this book we have only treated a limited set of expressions – the arithmetic expressions – in a special way. Much active research is concerned with extending this analysis to cover more general expressions [Burn 1991].

3.8 Mark 6: Adding data structures

In this section we extend the G-machine to deal with arbitrary data structures. As discussed in Chapter 1, two new Core-language constructs are required for programs involving data structures: *constructors*, `EConstr`; and *case expressions*, `ECase`. Our goal, in producing the Mark 6 machine, is to compile code for these expressions.

3.8.1 Overview

In Section 1.1.4 we saw that a constructor with tag `t` and arity `a` was represented as `Pack{t,a}` in the core language. For example, the usual list data type has two constructors: `Pack{1,0}` and `Pack{2,2}`. These correspond to Miranda's [] and (:) respectively.

A `case` expression, which has no direct counterpart in Miranda, is used to inspect the values held in a constructor. For example, we can write the length function for a list as:

```
length xs = case xs of
                <1>      -> 0;
                <2> y ys -> 1 + length ys
```

It is instructive to look at the way we execute the `case` expression

1. To evaluate the `case` expression, we first evaluate `xs` to WHNF.
2. Once this evaluation has occurred, we are able to tell which of the *alternatives* to take. The *tag* of the evaluated expression – which must be a structured data object – determines which alternative we take. In the above example, for `length`:
    - If the tag of the constructor for `xs` is `1` then the list is empty and we take the first branch. We therefore return `0`.
    - If the tag is `2`, then the list is non-empty. This time there are components of the constructor (`y` and `ys`). The length of the list is one more than the length of `ys`.

We will assume that whenever we attempt to dismantle a constructor it has been applied to the correct number of arguments. A constructor in this state is said to be *saturated*. As an example, in Section 1.1.3, `Cons` is defined to take two arguments, so it is saturated when it is applied to two expressions.

We also note that a Core-language program can now return a result that is a structured data object. The Mark 5 G-machine must be able to print the structured data object in a lazy fashion. Let us first consider what additions will need to be made to the data structures of the Mark 5 machine.

3.8.2 Data structures

It would be nice to allow the machine to return values which are not just numbers. We would like to be able to return values that consist of constructors. This will require us to evaluate the components of the structure recursively, and then return these values. To do this we need to add yet another component to the state: `gmOutput`. This will hold the result of the program.

```
> gmState == (gmOutput,          || Current output
>             gmCode,            || Current instruction stream
>             gmStack,           || Current stack
>             gmDump,            || Current dump
>             gmHeap,            || Heap of nodes
>             gmGlobals,         || Global addresses in heap
>             gmStats)           || Statistics
```

This component is defined to be a character string.

```
> gmOutput == [char]
```

We can write the access functions in the obvious way.

```
> getOutput :: gmState -> gmOutput
> getOutput (o, i, stack, dump, heap, globals, stats) = o


> putOutput :: gmOutput -> gmState -> gmState
> putOutput o' (o, i, stack, dump, heap, globals, stats)
>       = (o', i, stack, dump, heap, globals, stats)
```

Exercise 3.31. Make the appropriate changes to the remainder of the access functions.

To support constructor nodes in the heap, we augment the type `node` with `NConstr`; this takes a positive number which will represent a *tag*, and a list of *components* which we represent as the list of the addresses of the nodes in heap.

```
> node ::= NNum num             || Numbers
>          | NAp  addr addr     || Applications
>          | NGlobal num gmCode || Globals
>          | NInd addr          || Indirections
>          | NConstr num [addr] || Constructors
```

3.8.3 Printing the result

Because we have a new state component which we wish to display, we must redefine the function `showState`.

```
> showState :: gmState -> iseq
> showState s
>      = iConcat [showOutput s,                 iNewline,
>              showStack s,                  iNewline,
>              showDump s,                   iNewline,
>              showInstructions (getCode s), iNewline]
```

The showOutput function is easy, because the output component is already a string.

```
> showOutput :: gmState -> iseq
> showOutput s = iConcat [iStr "Output:\"", iStr (getOutput s), iStr "\""]
```

The only other change (apart from changing showInstruction for the new instruction set) occurs in showNode, because we have extended the data type to include constructor nodes.

```
> showNode :: gmState -> addr -> node -> iseq
> showNode s a (NNum n)      = iNum n
> showNode s a (NGlobal n g) = iConcat [iStr "Global ", iStr v]
>    where v = hd [n | (n,b) <- globals; a=b]
>          globals = getGlobals s
> showNode s a (NAp a1 a2)   = iConcat [iStr "Ap ",  iStr (showaddr a1),
>                                       iStr " ",    iStr (showaddr a2)]
> showNode s a (NInd a1)     = iConcat [iStr "Ind ", iStr (showaddr a1)]
> showNode s a (NConstr t as)
>  = iConcat [iStr "Cons ", iNum t, iStr " [",
>             iInterleave (iStr ", ") (map (iStr.showaddr) as), iStr "]"]
```

3.8.4 The instruction set

The new instruction set is now defined. It simply adds four new instructions to the Mark 4 machine.

```
> instruction ::= Slide num
>               | Alloc num
>               | Update num
>               | Pop num
>               | Unwind
>               | Pushglobal name
>               | Pushint num
>               | Push num
>               | Mkap
>               | Eval
>               | Add | Sub | Mul | Div
>               | Neg
>               | Eq | Ne | Le | Lt | Ge | Gt
>               | Cond gmCode gmCode
```

The four new instructions that are added to the machine are as follows:

```
>                    | Pack num num
>                    | Casejump [(num,gmCode)]
>                    | Split num
>                    | Print
```

Exercise 3.32. Extend `showInstruction` to match the new instruction set.

The `Pack` instruction is simple; it assumes that there are sufficient arguments on the stack to construct a *saturated constructor*. When there are, it proceeds to make a saturated constructor; if there are not enough arguments, then the instruction is undefined.

(3.30)

	$\mathbf{o}$	$\texttt{Pack}\ \mathbf{t}\ \mathbf{n} : \mathbf{i}$	$\mathbf{a}_1 : \ldots : \mathbf{a}_\mathbf{n} : \mathbf{s}$	$\mathbf{d}$	$\mathbf{h}$	$\mathbf{m}$
$\Longrightarrow$	$\mathbf{o}$	$\mathbf{i}$	$\mathbf{a} : \mathbf{s}$	$\mathbf{d}$	$\mathbf{h}[\mathbf{a} : \texttt{NConstr}\ \mathbf{t}[\mathbf{a}_1, \ldots, \mathbf{a}_\mathbf{n}]]$	$\mathbf{m}$

The transition rule for `Casejump` expects (a) that the node on top of the stack is in WHNF, and (b) that the node is a structured data object. Using the tag from this object we select one of the alternative instruction sequences, and the current instruction stream is then prefixed by the code for the particular alternative selected.

(3.31)

	$\mathbf{o}$	$\texttt{Casejump}\ [\ldots, \mathbf{t}\texttt{->}\mathbf{i}', \ldots] : \mathbf{i}$	$\mathbf{a} : \mathbf{s}$	$\mathbf{d}$	$\mathbf{h}[\mathbf{a} : \texttt{NConstr}\ \mathbf{t}\ \mathbf{ss}]$	$\mathbf{m}$
$\Longrightarrow$	$\mathbf{o}$	$\mathbf{i}' +\!\!+ \mathbf{i}$	$\mathbf{a} : \mathbf{s}$	$\mathbf{d}$	$\mathbf{h}$	$\mathbf{m}$

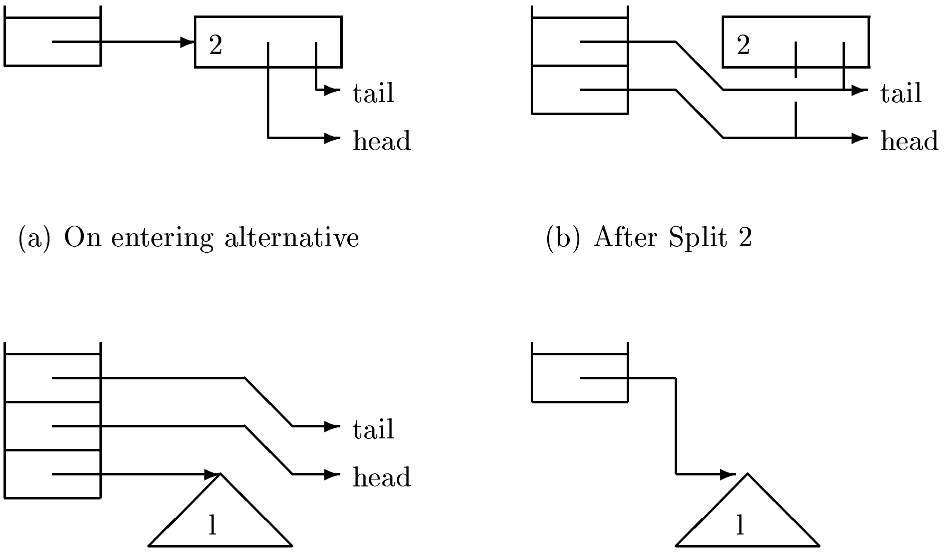
This is a simple way to specify a multiway jump and join. That is, by prefixing the current code $\mathbf{i}$ by the code for the alternative $\mathbf{i}'$, we achieve the effect of first running the code for the alternative and then resuming with whatever the remainder of the code for the main expression requires[3].

The code for each alternative begins with a `Split` $\mathbf{n}$ instruction and terminates with a `Slide` $\mathbf{n}$ instruction. The value of $\mathbf{n}$ is determined by the number of components in the constructor. The `Split` instruction is used to gain access to the components of a constructor.

Consider the code sequence generated for the `length` function:

```
[Push 0, Eval,
 Casejump [1 -> [Pushint 0]
           2 -> [Split 2, Push 1, Pushglobal "length", Mkap,
                 Eval, Pushint 1, Add, Slide 2]],
 Update 1,
 Pop 1,
 Unwind]
```

[3]It should be noted that code sequences using `Casejump` are not flat. We can, however, construct the flat code sequences we desire by labelling each alternative and jumping to the labelled code addresses. We have not done this as it unnecessarily complicates the code generation.



(a) On entering alternative (b) After Split 2

(c) After code for body (d) After Slide 2

Figure 3.13: Running compiled code for alternatives

The execution of this pattern is shown in Figure 3.13, where we see that the `Slide` and `Split` instructions are being used temporarily to extend the current set of local bindings. Assuming that the `length` function was applied to a non-nil node, when we execute the `Casejump` instruction, we take the alternative labelled 2. This is the initial diagram (a). The `Split 2` instruction 'unpacks' the constructor node onto the stack. This is shown in diagram (b). After completing the body of the alternative, i.e. the code sequence

```
[Push 1, Pushglobal "length", Mkap, Eval, Pushint 1, Add]
```

the length of the list argument to this call of `length` will be on top of the stack labelled `l` in diagram (c). To complete the execution we remove the pointers to `head` and `tail`; this is shown in diagram (d). The transition for `Split` is straightforward.

(3.32)

	$\mathbf{o}$	$\texttt{Split}\ \mathbf{n} : \mathbf{i}$	$\mathbf{a} : \mathbf{s}$	$\mathbf{d}$	$\mathbf{h}[\mathbf{a} : \texttt{NConstr}\ \mathbf{t}\ [\mathbf{a}_1, \ldots, \mathbf{a_n}]]$	$\mathbf{m}$
$\Longrightarrow$	$\mathbf{o}$	$\mathbf{i}$	$\mathbf{a}_1 : \ldots : \mathbf{a_n} : \mathbf{s}$	$\mathbf{d}$	$\mathbf{h}$	$\mathbf{m}$

Finally, we describe the transitions for `Print`. There are two transitions for `Print`; one each for constructors and numbers.

(3.33)

	$\mathbf{o}$	$\texttt{Print} : \mathbf{i}$	$\mathbf{a} : \mathbf{s}$	$\mathbf{d}$	$\mathbf{h}[\mathbf{a} : \texttt{NNum}\ \mathbf{n}]$	$\mathbf{m}$
$\Longrightarrow$	$\mathbf{o} +\!\!+ [\mathbf{n}]$	$\mathbf{i}$	$\mathbf{s}$	$\mathbf{d}$	$\mathbf{h}$	$\mathbf{m}$

The rule for constructors is more complex, as it must arrange to print each component of the constructor. For simplicity we will only print out the components.

(3.34)

	$\mathbf{o}$	$\texttt{Print} : \mathbf{i}$	$\mathbf{a} : \mathbf{s}$	$\mathbf{d}$	$\mathbf{h}[\mathbf{a} : \texttt{NConstr}\ \mathbf{t}\ [\mathbf{a}_1, \ldots, \mathbf{a_n}]]$	$\mathbf{m}$
$\Longrightarrow$	$\mathbf{o}$	$\mathbf{i}' +\!\!+ \mathbf{i}$	$\mathbf{a}_1 : \ldots : \mathbf{a_n} : \mathbf{s}$	$\mathbf{d}$	$\mathbf{h}$	$\mathbf{m}$

$\mathcal{E}[\![\mathbf{e}]\!]\ \rho$ compiles code that evaluates an expression $\mathbf{e}$ to WHNF in environment ρ, leaving a pointer to the expression on top of the stack.

$$\begin{array}{rcl}
\mathcal{E}[\![\texttt{case}\ \mathbf{e}\ \texttt{of}\ \mathbf{alts}]\!]\ \rho & = & \mathcal{E}[\![\mathbf{e}]\!]\ \rho \mathbin{+\!\!+} [\texttt{Casejump}\ \mathcal{D}[\![\mathbf{alts}]\!]\ \rho] \\
\mathcal{E}[\![\texttt{Pack\{}\mathbf{t}\texttt{,}\mathbf{a}\texttt{\}}\ \mathbf{e}_1\ \ldots\ \mathbf{e}_\mathbf{a}]\!]\ \rho & = & \mathcal{C}[\![\mathbf{e}_\mathbf{a}]\!]^{+0}\ \rho \mathbin{+\!\!+} \ldots \mathcal{C}[\![\mathbf{e}_1]\!]^{+(\mathbf{a}-1)}\ \rho \mathbin{+\!\!+} [\texttt{Pack}\ \mathbf{t}\ \mathbf{a}]
\end{array}$$

$\mathcal{C}[\![\mathbf{e}]\!]\ \rho$ generates code which constructs the graph of $\mathbf{e}$ in environment ρ, leaving a pointer to it on top of the stack.

$$\mathcal{C}[\![\texttt{Pack\{}\mathbf{t}\texttt{,}\mathbf{a}\texttt{\}}\ \mathbf{e}_1\ \ldots\ \mathbf{e}_\mathbf{a}]\!]\ \rho = \mathcal{C}[\![\mathbf{e}_\mathbf{a}]\!]^{+0}\ \rho \mathbin{+\!\!+} \ldots \mathcal{C}[\![\mathbf{e}_1]\!]^{+(\mathbf{a}-1)}\ \rho \mathbin{+\!\!+} [\texttt{Pack}\ \mathbf{t}\ \mathbf{a}]$$

$\mathcal{D}[\![\mathbf{alts}]\!]\ \rho$ compiles the code for the alternatives in a `case` expression.

$$\mathcal{D}[\![\mathbf{alt}_1\ \ldots\ \mathbf{alt}_\mathbf{n}]\!]\ \rho = [\mathcal{A}[\![\mathbf{alt}_1]\!]\ \rho, \ldots, \mathcal{A}[\![\mathbf{alt}_\mathbf{n}]\!]\ \rho]$$

$\mathcal{A}[\![\mathbf{alt}]\!]\ \rho$ compiles the code for an alternative in a `case` expression.

$$\begin{array}{rcl}
\mathcal{A}[\![\texttt{<t>}\ \mathbf{x}_1 \ldots \mathbf{x}_\mathbf{n}\ \texttt{->}\ \mathbf{body}]\!]\ \rho & = & \mathbf{t}\ \texttt{->}\ [\texttt{Split}\ \mathbf{n}] \mathbin{+\!\!+} \mathcal{E}[\![\mathbf{body}]\!]\ \rho' \mathbin{+\!\!+} [\texttt{Slide}\ \mathbf{n}] \\
& & \text{where } \rho' = \rho^{+\mathbf{n}}[\mathbf{x}_1 \mapsto 0 \ldots \mathbf{x}_\mathbf{n} \mapsto \mathbf{n}-1]
\end{array}$$

Figure 3.14: Compilation schemes for `case` expressions

The code $\mathbf{i}'$ is simply:

$$[\underbrace{\texttt{Eval, Print}, \ldots, \texttt{Eval, Print}}_{\mathbf{n}}].$$

Exercise 3.33. Implement the new transitions and modify the `dispatch` function.

3.8.5 The compiler

In Figure 3.14 the new cases for the $\mathcal{E}$ and $\mathcal{C}$ compilation schemes are presented. They require auxiliary compilation schemes, $\mathcal{D}$ and $\mathcal{A}$, to deal with the alternatives that may be selected in a `case` expression. The function `compileAlts` (corresponding to the $\mathcal{D}$ scheme) compiles a list of alternatives, using the current environment, and produces a list of tagged code sequences. It also uses the `comp` argument (which corresponds to $\mathcal{A}$) to compile the body of each of the alternatives. For the moment this argument will always be `compileE'`.

```
> compileAlts :: (num -> gmCompiler) || compiler for alternative bodies
>                -> [coreAlt]        || the list of alternatives
>                -> gmEnvironment    || the current environment
>                -> [(num, gmCode)]  || list of alternative code sequences
> compileAlts comp alts env
```

```
>     = [(tag, comp (#names) body (zip2 names [0..] ++ argOffset (#names) env))
>           | (tag, names, body) <- alts]
```

The `compileE'` scheme is a small modification to the `compileE` scheme. It simply places a `Split` and `Slide` around the code generated by the ordinary `compileE` scheme.

```
> compileE' :: num -> gmCompiler
> compileE' offset expr env
>     = [Split offset] ++ compileE expr env ++ [Slide offset]
```

Exercise 3.34. Make the relevant changes to `compile`, and modify `initialCode` to have a final `Print` instruction.

Exercise 3.35. Add the new cases to the compiler functions `compileE` and `compileC`.

Exercise 3.36. What changes are required to print out the output in 'structured form'. By this we mean placing the constructors and parentheses into the output component `gmOutput`, as well as integers.

3.8.6 Using the new boolean representation in comparisons

In this section we show how the Mark 6 machine we have constructed, can be modified to use our new representation of booleans. We first observe that we can implement the booleans as structured data objects; with `True` and `False` being represented as constructors of zero arity and tags 2 and 1 respectively.

How do we implement conditionals? This can be done by adding a new definition to the program for `if`. It returns either its second or third argument, depending on the first argument.

```
if c t f = case c of
                <1> -> f;
                <2> -> t
```

The first change we require lies in the comparison operations. These have the following generic transition rule.

$$(3.35)\quad \boxed{\begin{array}{lllllll} & \mathbf{o} & \odot : \mathbf{i} & \mathbf{a}_0 : \mathbf{a}_1 : \mathbf{s} & \mathbf{d} & \mathbf{h}[\mathbf{a}_0 : \texttt{NNum}\ \mathbf{n}_0,\ \mathbf{a}_1 : \texttt{NNum}\ \mathbf{n}_1] & \mathbf{m} \\ \Longrightarrow & \mathbf{o} & \mathbf{i} & \mathbf{a} : \mathbf{s} & \mathbf{d} & \mathbf{h}[\mathbf{a} : \texttt{Constr}\ (\mathbf{n}_0 \odot \mathbf{n}_1)\ []] & \mathbf{m} \end{array}}$$

For example, in the `Eq` instruction, we replace $\odot$ with a function that returns `2` (the tag for `True`) if the two numbers $\mathbf{n}_0$ and $\mathbf{n}_1$ are the same, and `1` (the tag for `False`) otherwise.

We can implement the transitions quickly, by reusing some of the code we developed for the Mark 4 machine. In Section 3.6.3 we saw how to represent some generic arithmetic and comparison operators. In fact, because of the way in which we structured the definition of the `comparison` function, we can almost immediately use our new representation of booleans.

The boxing function `boxBoolean` takes a comparison operation and a state in which there are two integers on top of the stack. It returns the new state in which there is the boolean result of comparing the two integers on top of the stack.

```
> boxBoolean :: bool -> gmState -> gmState
> boxBoolean b state
> = putStack (a: getStack state) (putHeap h' state)
>   where (h',a) = hAlloc (getHeap state) (NConstr b' [])
>         b' = 2, b                   || 2 is tag of True
>            = 1, otherwise           || 1 is tag of False
```

Exercise 3.37. Run some example programs from Appendix B; for example try the factorial program:

```
fac n = if (n==0) 1 (n * fac (n-1))
```

3.8.7 Extending the language accepted

As astute readers might have noticed, there are some legal expressions involving `ECase` and `EConstr` for which our compiler will fail. The legal expressions that we cannot compile fall into two classes:

1. Occurrences of `ECase` in non-strict contexts; i.e. in expressions compiled by the $\mathcal{C}$ scheme.
2. Occurrences of `EConstr` in expressions where it is applied to too few arguments.

Both problems can be solved by using program transformation techniques. The solution for `ECase` is to make the offending expressions into supercombinators which are then applied to their free variables. For example, the program:

```
f x = Pack{2,2} (case x of <1> -> 1; <2> -> 2) Pack{1,0}
```

can be transformed into the equivalent program:

```
f x = Pack{2,2} (g x) Pack{1,0}
g x = case x of <1> -> 1; <2> -> 2
```

The trick for `EConstr` is to create a supercombinator for each constructor; this will be generated with enough free variables to saturate the constructor. Here is an example.

```
prefix p xs = map (Pack{2,2} p) xs
```

This is transformed to:

```
prefix p xs = map (f p) xs
f p x = Pack{2,2} p x
```

Another way to solve this problem is to modify the `Pushglobal` instruction, so that it will work for functions with names of the form: '`Pack{t,a}`'. We can then simply look for constructor functions, such as `f` in the example above, in the `globals` component of the state. If the function is not already present, we can create a new global node to associate with the function because the node has a particularly simple structure.

```
NGlobal a [Pack t a, Update 0, Unwind]
```

The new transitions are, firstly, if the function exists already:

(3.36)

	$\mathbf{o}$	Pushglobal Pack{$\mathbf{t}$,$\mathbf{n}$} : $\mathbf{i}$	$\mathbf{s}$	$\mathbf{d}$	$\mathbf{h}$	$\mathbf{m}$[Pack{$\mathbf{t}$,$\mathbf{n}$} : $\mathbf{a}$]
$\Longrightarrow$	$\mathbf{o}$	$\mathbf{i}$	$\mathbf{a} : \mathbf{s}$	$\mathbf{d}$	$\mathbf{h}$	$\mathbf{m}$

and secondly when it does not already exist:

(3.37)

	$\mathbf{o}$	Pushglobal Pack{$\mathbf{t}$,$\mathbf{n}$} : $\mathbf{i}$	$\mathbf{s}$	$\mathbf{d}$	$\mathbf{h}$	$\mathbf{m}$
$\Longrightarrow$	$\mathbf{o}$	$\mathbf{i}$	$\mathbf{a} : \mathbf{s}$	$\mathbf{d}$	$\mathbf{h}[\mathbf{a} :$ gNode$_{\mathbf{t},\mathbf{n}}]$	$\mathbf{m}$[Pack{$\mathbf{t}$,$\mathbf{n}$} : $\mathbf{a}$]

where gNode$_{\mathbf{t},\mathbf{n}}$ is

NGlobal $\mathbf{n}$ [Pack $\mathbf{t}$ $\mathbf{n}$, Update 0, Unwind]

Our compiler can then generate code for expressions with unsaturated constructor nodes directly. It does this by generating the following code for unsaturated constructors.

$$\mathcal{C}[\![\texttt{Pack\{t,a\}}]\!]\ \rho = [\texttt{Pushglobal "Pack\{t,a\}"}]$$

Exercise 3.38. Implement the extensions to the `pushglobal` function for the `Pushglobal` instruction and modify the compiler.

3.9 Mark 7: Further improvements

Let us consider again the example program we saw when we developed the Mark 5 compiler.

```
main = 3+4*5
```

This generates the following code when we use the Mark 6 compiler:

```
[Pushint 5, Pushint 4, Mul, Pushint 3, Add]
```

When executed this code will use five heap nodes. Is it possible to reduce this still further?

The answer is yes. We can reduce the number of heap accesses for arithmetic further by using a stack of numbers to represent intermediate values in the computation. In the Mark 7 machine these values are held in a new state component called the *V-stack*. The problem is that placing numbers into the heap or extracting them is an expensive operation on a real machine. It is much more efficient to use the machine's register set or stack. In the Mark 7 G-machine we will use a stack; this means that we do not have to worry about running out of registers.

The new code for the program

```
main = 3+4*5
```

is very similar to that which we previously generated:

```
[Pushbasic 5, Pushbasic 4, Mul, Pushbasic 3, Add, Mkint]
```

The first instruction `Pushbasic 5` places `5` on top of the V-stack. Next we push `4` onto the V-stack, following this by a multiplication. This instruction now expects its arguments to be in the V-stack. It will place the answer into the V-stack as well. The next two instructions add `3` to the value on top of the V-stack. The final instruction, `Mkint`, takes the value on top of the V-stack and places it into a number node in the heap, leaving a pointer to this new node on top of the S-stack.

3.9.1 Executing the factorial function using the V-stack

We begin an investigation into the Mark 7 machine, by way of an example. We will be looking at the execution of the factorial function, defined as follows:

```
fac n = if (n==0) 1 (n * fac (n-1))
```

Using the Mark 7 compiler, we will generate the following code sequence for the body of the supercombinator.

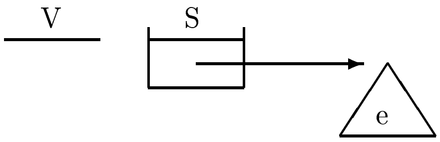
```
[Pushbasic 0, Push 0, Eval, Get, Eq,
 Cond [Pushint 1, Update 1, Pop 1, Unwind]
      [Pushint 1, Push 1, Pushglobal "-", Mkap, Mkap, Pushglobal "fac", Mkap,
       Eval, Get, Push 0, Eval, Get, Mul, Mkint, Update 1, Pop 1, Unwind]
```

When this code commences execution, the V-stack is empty, and there is one item on the ordinary stack. We will call the latter the S-stack from now on to distinguish the two sorts of stack. In Figure 3.15 – starting with diagram (a) – we see the initial state in which a pointer to the argument to `fac` is on top of the S-stack. In diagram (b) we see what happens when a `Pushbasic` instruction is executed: an integer is pushed onto the V-stack. Diagram (c) shows that the argument to `fac` has now been evaluated, whilst in diagram (d) we see the effect of a `Get` instruction. It has extracted the value from the node in the heap, and placed it into the V-stack.

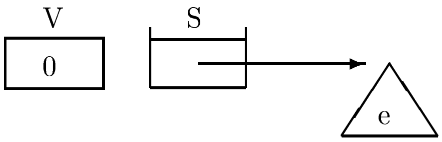
In diagram (e) (Figure 3.16) we see the state after an `Eq` instruction has executed. It has compared the two items in the V stack, and discovered that they are not equal. The Mark 7 G-machine represents the boolean value `False` by `1` in the V-stack. In diagram (f), the `Cond` instruction has inspected this value, and used it to select which branch to execute.

In diagram (g) (Figure 3.17) the state after the construction and evaluation of `fac (1-1)` is shown. The next instruction is `Get` which fetches the newly evaluated value into the V stack. Diagram (i) shows that we evaluate and fetch the value from the node `1` into the V-stack. In diagram (j) a `Mul` instruction has multiplied the two values in the V-stack, placing the result back there. In diagram (k) a `Mkint` instruction has moved this result from the V-stack to the heap, recording the address of the newly created node on the S-stack.

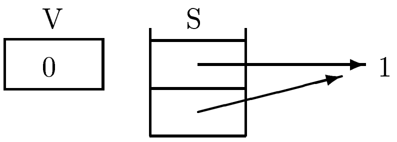
In a machine where there is a performance penalty for creating and accessing objects in the heap – compared with keeping the objects in a stack – we expect the use of the V-stack to be



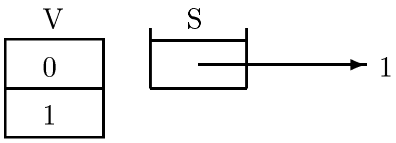
(a) State on entry to fac



(b) After Pushbasic 0

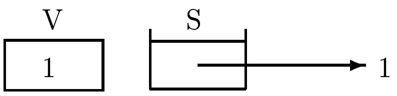


(c) After Push 0, Eval

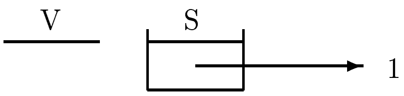


(d) After Get

Figure 3.15: Mark 7 machine running `fac`

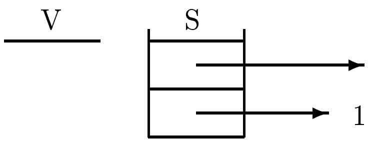


(e) After Eq

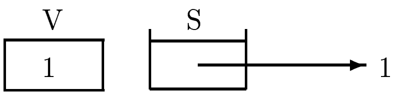


(f) After Cond

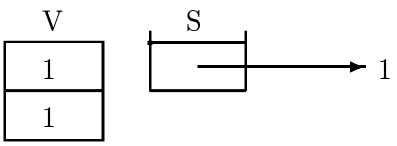
Figure 3.16: Mark 7 machine running `fac`



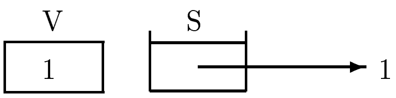
(g) After Pushint 1, ... Mkap, Eval



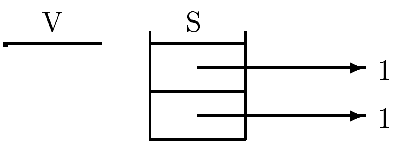
(h) After Get



(i) After Push 0, Eval, Get



(j) After Mul



(k) After Mkint

Figure 3.17: Mark 7 machine running `fac`

an improvement. Having seen how the V-stack works, we now make small modifications to the G-machine to implement the Mark 7 machine.

3.9.2 Data structures

The use of the V-stack requires that each G-machine state has a new state component `gmVStack` added to its state.

```
> gmState == (gmOutput,           || Current output
>             gmCode,             || Current instruction stream
>             gmStack,            || Current stack
>             gmDump,             || Current dump
>             gmVStack,           || Current V-stack
>             gmHeap,             || Heap of nodes
>             gmGlobals,          || Global addresses in heap
>             gmStats)            || Statistics
```

As we have already stated this new component behaves as a stack of numbers.

```
> gmVStack == [num]
```

We add access functions for this component.

```
> getVStack :: gmState -> gmVStack
> getVStack (o, i, stack, dump, vstack, heap, globals, stats) = vstack


> putVStack :: gmVStack -> gmState -> gmState
> putVStack vstack' (o, i, stack, dump, vstack, heap, globals, stats)
>      = (o, i, stack, dump, vstack', heap, globals, stats)
```

Exercise 3.39. Make the relevant changes to the other access functions.

Displaying the states

The function `showState` is changed so that it prints out the V-stack component.

```
> showState :: gmState -> iseq
> showState s
>      = iConcat [showOutput s,                 iNewline,
>            showStack s,                  iNewline,
>            showDump s,                   iNewline,
>            showVStack s,                 iNewline,
>            showInstructions (getCode s), iNewline]
```

To do this we define the function `showVStack`.

```
> showVStack :: gmState -> iseq
> showVStack s
>      = iConcat [iStr "Vstack:[",
>            iInterleave (iStr ", ") (map iNum (getVStack s)),
>            iStr "]"]
```

3.9.3 Instruction set

An obvious first requirement is that each of the arithmetic transitions will now have to be modified to get their values from, and return their results to, the V-stack instead of the ordinary stack. Let us first deal with the case of dyadic primitives. The generic transition for the operation $\odot$ is given below. It takes two arguments from the V-stack and places the result of the operation $\odot$ back onto the V-stack.

$$(3.38)\quad \begin{array}{|lrlllrll|} \hline & \mathbf{o} & \odot : \mathbf{i} & \mathbf{s} & \mathbf{d} & \mathbf{n}_0 : \mathbf{n}_1 : \mathbf{v} & \mathbf{h} & \mathbf{m} \\ \Longrightarrow & \mathbf{o} & \quad\ \mathbf{i} & \mathbf{s} & \mathbf{d} & \mathbf{n}_0 \odot \mathbf{n}_1 : \mathbf{v} & \mathbf{h} & \mathbf{m} \\ \hline \end{array}$$

The `Neg` instruction now has the following transition: it simply replaces the number on top of the V-stack with its negation.

$$(3.39)\quad \begin{array}{|lrrllrll|} \hline & \mathbf{o} & \texttt{Neg} : \mathbf{i} & \mathbf{s} & \mathbf{d} & \mathbf{n} : \mathbf{v} & \mathbf{h} & \mathbf{m} \\ \Longrightarrow & \mathbf{o} & \mathbf{i} & \mathbf{s} & \mathbf{d} & (-\mathbf{n}) : \mathbf{v} & \mathbf{h} & \mathbf{m} \\ \hline \end{array}$$

We also need instructions to move values between the heap and the V-stack. We begin with `Pushbasic`, which pushes an integer $\mathbf{n}$ onto the V-stack.

$$(3.40)\quad \begin{array}{|lrrllrll|} \hline & \mathbf{o} & \texttt{Pushbasic}\ \mathbf{n} : \mathbf{i} & \mathbf{s} & \mathbf{d} & \mathbf{v} & \mathbf{h} & \mathbf{m} \\ \Longrightarrow & \mathbf{o} & \mathbf{i} & \mathbf{s} & \mathbf{d} & \mathbf{n} : \mathbf{v} & \mathbf{h} & \mathbf{m} \\ \hline \end{array}$$

To move a value from the V-stack to the heap, we use two instructions: `Mkbool` and `Mkint`. These treat the integer on top of the V-stack as booleans and integers respectively. We begin with `Mkbool`.

$$(3.41)\quad \begin{array}{|lrrrlrll|} \hline & \mathbf{o} & \texttt{Mkbool} : \mathbf{i} & \mathbf{s} & \mathbf{d} & \mathbf{t} : \mathbf{v} & \mathbf{h} & \mathbf{m} \\ \Longrightarrow & \mathbf{o} & \mathbf{i} & \mathbf{a} : \mathbf{s} & \mathbf{d} & \mathbf{v} & \mathbf{h}[\mathbf{a} : \texttt{NConstr}\ \mathbf{t}\ []] & \mathbf{m} \\ \hline \end{array}$$

The transition for `Mkint` is similar, except that it creates a new integer node in the heap.

$$(3.42)\quad \begin{array}{|lrrrlrll|} \hline & \mathbf{o} & \texttt{Mkint} : \mathbf{i} & \mathbf{s} & \mathbf{d} & \mathbf{n} : \mathbf{v} & \mathbf{h} & \mathbf{m} \\ \Longrightarrow & \mathbf{o} & \mathbf{i} & \mathbf{a} : \mathbf{s} & \mathbf{d} & \mathbf{v} & \mathbf{h}[\mathbf{a} : \texttt{NNum}\ \mathbf{n}] & \mathbf{m} \\ \hline \end{array}$$

To perform the inverse operation we use `Get`. This is specified by two transitions. In the first we see how `Get` treats a boolean on top of the stack.

$$
\begin{array}{|rlrrrrlr|}
\hline
 & o & \texttt{Get} : i & a : s & d & v & h[a : \texttt{NConstr}\ t\ []] & m \\
\Longrightarrow & o & i & s & d & t : v & h & m \\
\hline
\end{array} \qquad (3.43)
$$

In the second, we see how `Get` treats a number.

$$
\begin{array}{|rlrrrrlr|}
\hline
 & o & \texttt{Get} : i & a : s & d & v & h[a : \texttt{NNum}\ n] & m \\
\Longrightarrow & o & i & s & d & n : v & h & m \\
\hline
\end{array} \qquad (3.44)
$$

Finally, to make use of booleans on the V-stack we use a simplified `Casejump` instruction that inspects the V-stack to determine which instruction stream to use. This new instruction is called `Cond`, and is specified by the following two transitions. In the first – with the value on top of the V-stack being true – we select the first code sequence: **t**.

$$
\begin{array}{|rlrrrrrr|}
\hline
 & o & \texttt{Cond}\ t\ f : i & s & d & 2 : v & h & m \\
\Longrightarrow & o & t \mathbin{+\!\!+} i & s & d & v & h & m \\
\hline
\end{array} \qquad (3.45)
$$

In the second transition we see that with a false value on top of the V-stack `Cond` selects its second code sequence: **f**.

$$
\begin{array}{|rlrrrrrr|}
\hline
 & o & \texttt{Cond}\ t\ f : i & s & d & 1 : v & h & m \\
\Longrightarrow & o & f \mathbin{+\!\!+} i & s & d & v & h & m \\
\hline
\end{array} \qquad (3.46)
$$

Exercise 3.40. Extend the instruction data type, redefine the `showInstruction` function, implement the new instruction transitions and modify the `dispatch` function.

We now consider the – rather extensive – modifications to the compiler.

3.9.4 The compiler

Because of the extra state component, the `compile` function must initialise the V-stack component to be empty.

```
> compile :: coreProgram -> gmState
> compile program
>      = ([], initialCode, [], [], [], heap, globals, statInitial)
>      where (heap, globals) = buildInitialHeap program
```

Strictly speaking, this is all that is necessary to make the machine work, but we have introduced the V-stack so that we can compile arithmetic functions 'in-line', so this is what we intend our code to do.

```
> buildInitialHeap :: coreProgram -> (gmHeap, gmGlobals)
> buildInitialHeap program
>      = mapAccuml allocateSc hInitial compiled
>      where compiled = map compileSc (preludeDefs ++ program ++ primitives)
```

Because of the changes to the transitions for the primitive instruction, we must change the code for each compiled primitive. Instead of hand compiling this code – as we did for the Mark 6 machine – we can instead give this job to the compiler. This of course relies on the fact that the compiler is clever enough to optimise the code it produces, otherwise we never generate any `Add` instructions!

```
> primitives :: [(name,[name],coreExpr)]
> primitives
>       =       [("+", ["x","y"], (EAp (EAp (EVar "+") (EVar "x")) (EVar "y"))),
>                ("-", ["x","y"], (EAp (EAp (EVar "-") (EVar "x")) (EVar "y"))),
>                ("*", ["x","y"], (EAp (EAp (EVar "*") (EVar "x")) (EVar "y"))),
>                ("/", ["x","y"], (EAp (EAp (EVar "/") (EVar "x")) (EVar "y"))),
```

We also need to add the negation function.

```
>                ("negate", ["x"], (EAp (EVar "negate") (EVar "x"))),
```

Comparison functions are almost identical to the dyadic arithmetic functions.

```
>                ("==", ["x","y"], (EAp (EAp (EVar "==") (EVar "x")) (EVar "y"))),
>                ("~=", ["x","y"], (EAp (EAp (EVar "~=") (EVar "x")) (EVar "y"))),
>                (">=", ["x","y"], (EAp (EAp (EVar ">=") (EVar "x")) (EVar "y"))),
>                (">",  ["x","y"], (EAp (EAp (EVar ">")  (EVar "x")) (EVar "y"))),
>                ("<=", ["x","y"], (EAp (EAp (EVar "<=") (EVar "x")) (EVar "y"))),
>                ("<",  ["x","y"], (EAp (EAp (EVar "<")  (EVar "x")) (EVar "y"))),
```

Finally, we ought to include the conditional function, and some supercombinators to represent boolean values.

```
>                ("if",  ["c","t","f"],
>                       (EAp (EAp (EAp (EVar "if") (EVar "c")) (EVar "t")) (EVar "f"))),
>                ("True",  [], (EConstr 2 0)),
>                ("False", [], (EConstr 1 0))]
```

The $\mathcal{B}$ compilation scheme

The $\mathcal{B}$ scheme, shown in Figure 3.18, constitutes another type of context. To be compiled by the $\mathcal{B}$ scheme, an expression must not only be known to need evaluation to WHNF, it must also be an expression of type integer or boolean. The following expressions will propagate through the $\mathcal{B}$ scheme:

- If `let(rec)` Δ `in` $\mathbf{e}$ occurs in a $\mathcal{B}$-strict context then the expression $\mathbf{e}$ is also in a $\mathcal{B}$-strict context.
- If the expression `if` $\mathbf{e}_0$ $\mathbf{e}_1$ $\mathbf{e}_2$ occurs in a $\mathcal{B}$-strict context then the expressions $\mathbf{e}_0$, $\mathbf{e}_1$ and $\mathbf{e}_2$ also occur in $\mathcal{B}$-strict contexts.

$\mathcal{B}[\![\mathbf{e}]\!]\ \rho$ compiles code that evaluates an expression $\mathbf{e}$ to WHNF, in an environment ρ, leaving the result on the V stack.

$$
\begin{array}{rcl}
\mathcal{B}[\![\mathbf{i}]\!]\ \rho & = & [\texttt{Pushbasic}\ \mathbf{i}] \\
\multicolumn{3}{l}{\mathcal{B}[\![\texttt{let}\ \mathbf{x}_1\texttt{=}\mathbf{e}_1\texttt{;}\ \ldots\texttt{;}\ \mathbf{x_n}\texttt{=}\mathbf{e_n}\ \texttt{in}\ \mathbf{e}]\!]\ \rho} \\
 & = & \mathcal{C}[\![\mathbf{e}_1]\!]\ \rho^{+0} \mathbin{+\!\!+} \ldots \mathbin{+\!\!+} \\
 & & \mathcal{C}[\![\mathbf{e_n}]\!]\ \rho^{+(\mathbf{n}-1)} \mathbin{+\!\!+} \\
 & & \mathcal{B}[\![\mathbf{e}]\!]\ \rho' \mathbin{+\!\!+} [\texttt{Pop}\ \mathbf{n}] \\
 & & \qquad \text{where } \rho' = \rho^{+\mathbf{n}}[\mathbf{x}_1 \mapsto \mathbf{n}-1, \ldots, \mathbf{x_n} \mapsto 0] \\
\multicolumn{3}{l}{\mathcal{B}[\![\texttt{letrec}\ \mathbf{x}_1\texttt{=}\mathbf{e}_1\texttt{;}\ \ldots\texttt{;}\ \mathbf{x_n}\texttt{=}\mathbf{e_n}\ \texttt{in}\ \mathbf{e}]\!]\ \rho} \\
 & = & [\texttt{Alloc}\ \mathbf{n}] \mathbin{+\!\!+} \\
 & & \mathcal{C}[\![\mathbf{e}_1]\!]\ \rho' \mathbin{+\!\!+} [\texttt{Update}\ \mathbf{n}-1] \mathbin{+\!\!+} \ldots \mathbin{+\!\!+} \\
 & & \mathcal{C}[\![\mathbf{e_n}]\!]\ \rho' \mathbin{+\!\!+} [\texttt{Update}\ 0] \mathbin{+\!\!+} \\
 & & \mathcal{B}[\![\mathbf{e}]\!]\ \rho' \mathbin{+\!\!+} [\texttt{Pop}\ \mathbf{n}] \\
 & & \qquad \text{where } \rho' = \rho^{+\mathbf{n}}[\mathbf{x}_1 \mapsto \mathbf{n}-1, \ldots, \mathbf{x_n} \mapsto 0] \\
\mathcal{B}[\![\mathbf{e}_0\ \texttt{+}\ \mathbf{e}_1]\!]\ \rho & = & \mathcal{B}[\![\mathbf{e}_1]\!]\ \rho \mathbin{+\!\!+} \mathcal{B}[\![\mathbf{e}_0]\!]\ \rho \mathbin{+\!\!+} [\texttt{Add}] \\
 & & \qquad \text{And similarly for other arithmetic expressions} \\
\mathcal{B}[\![\mathbf{e}_0\ \texttt{==}\ \mathbf{e}_1]\!]\ \rho & = & \mathcal{B}[\![\mathbf{e}_1]\!]\ \rho \mathbin{+\!\!+} \mathcal{B}[\![\mathbf{e}_0]\!]\ \rho \mathbin{+\!\!+} [\texttt{Eq}] \\
 & & \qquad \text{And similarly for other comparison expressions} \\
\mathcal{B}[\![\texttt{negate}\ \mathbf{e}]\!]\ \rho & = & \mathcal{B}[\![\mathbf{e}]\!]\ \rho \mathbin{+\!\!+} [\texttt{Neg}] \\
\mathcal{B}[\![\texttt{if}\ \mathbf{e}_0\ \mathbf{e}_1\ \mathbf{e}_2]\!]\ \rho & = & \mathcal{B}[\![\mathbf{e}_0]\!]\ \rho \mathbin{+\!\!+} [\texttt{Cond}\ (\mathcal{B}[\![\mathbf{e}_1]\!]\ \rho)\ (\mathcal{B}[\![\mathbf{e}_2]\!]\ \rho)] \\
\mathcal{B}[\![\mathbf{e}]\!]\ \rho & = & \mathcal{E}[\![\mathbf{e}]\!]\ \rho \mathbin{+\!\!+} [\texttt{Get}] \qquad \text{the default case}
\end{array}
$$

Figure 3.18: The $\mathcal{B}$ compilation scheme

- If $\mathbf{e}_0 \odot \mathbf{e}_1$ occurs in a $\mathcal{B}$-strict context, with $\odot$ being a comparison or arithmetic operator, then the expressions $\mathbf{e}_0$ and $\mathbf{e}_1$ also occur in $\mathcal{B}$-strict contexts.
- If `negate` $\mathbf{e}$ occurs in a $\mathcal{B}$-strict context then so does the expression $\mathbf{e}$.

If we cannot recognise any of the special cases of expression, then the compiled code will evaluate the expression using the $\mathcal{E}$ scheme, and then perform a `Get` instruction. The `Get` instruction unboxes the value left on top of the stack by the $\mathcal{E}$ scheme and moves it to the V-stack.

This has left unspecified how we know that an expression is initially in a $\mathcal{B}$-strict context. The usual situation is that we generate a $\mathcal{B}$-strict context from the usual strict context: $\mathcal{E}$, with the additional knowledge that the value is of type integer or boolean.

Exercise 3.41. Implement the $\mathcal{B}$ compiler scheme.

The $\mathcal{E}$ compilation scheme

The $\mathcal{E}$ scheme – defined in Figure 3.19 – specifies that we should give special treatment to arithmetic and comparison function calls. It differs from the version we used for the Mark 6 machine because we make use of the $\mathcal{B}$ scheme to perform the arithmetic and comparison operations. Again, if there are no special cases, then we must use a default method to compile the expression.

This is simply to build the graph, using the $\mathcal{C}$ scheme, and then place an `Eval` instruction in the code stream. This will ensure that the graph is evaluated to WHNF.

Exercise 3.42. Implement the new $\mathcal{E}$ compiler scheme.

The $\mathcal{R}$ compilation scheme

We also take this opportunity to improve the $\mathcal{R}$ scheme. Firstly, we wish to create opportunities for the $\mathcal{B}$ scheme to be used. We are also attempting to reduce the number of instructions that are executed at the end of a function's code sequence. The new compilation schemes for the $\mathcal{R}$ scheme are given in Figure 3.20. It has been expanded from the version we used in the Mark 6 machine; in that it now works like a *context*. We refer to this context as $\mathcal{R}$-strict. An expression being compiled in a $\mathcal{R}$-strict context will be evaluated to WHNF, and it will then be used to overwrite the current redex. It obeys the following rules of propagation.

- The expression in the body of a supercombinator definition is in an $\mathcal{R}$-strict context.
- If `let(rec)` Δ `in` **e** occurs in an $\mathcal{R}$-strict context then the expression **e** is also in an $\mathcal{R}$-strict context.
- If the expression `if` $\mathbf{e}_0$ $\mathbf{e}_1$ $\mathbf{e}_2$ occurs in an $\mathcal{R}$-strict context then the expressions $\mathbf{e}_1$ and $\mathbf{e}_2$ are also in an $\mathcal{R}$-strict context. (The expression $\mathbf{e}_0$ will now appear in a $\mathcal{B}$-strict context.)
- If `case` **e** `of` **alts** occurs in an $\mathcal{R}$-strict context then the expression **e** is in a strict context. Furthermore, the expression part of each alternative will occur in an $\mathcal{R}$-strict context.

Exercise 3.43. Implement the $\mathcal{R}$ scheme in Figure 3.20. Note that you can use the generality of the `compileAlts` function to implement both the $\mathcal{A}_\mathcal{R}$ and $\mathcal{A}_\mathcal{E}$ schemes.

One point worth noting about the Mark 7 machine is that we did not define the `Eval` instruction to save the current V-stack on the dump. This is in contrast to the G-machine described in [Peyton Jones 1987]. Whether you do this is really a matter of taste in the abstract machines we have produced in this book. In compiling code for a real machine we would be likely to try to minimise the number of stacks. When this is the case, we will wish to use the following alternative transition for `Eval`.

(3.47)

	$\mathbf{o}$	`Eval` : $\mathbf{i}$	$\mathbf{a} : \mathbf{s}$	$\mathbf{d}$	$\mathbf{v}$	$\mathbf{h}$	$\mathbf{m}$
$\Longrightarrow$	$\mathbf{o}$	`[Unwind]`	$[\mathbf{a}]$	$\langle \mathbf{i}, \mathbf{s}, \mathbf{v}\rangle : \mathbf{d}$	$[]$	$\mathbf{h}$	$\mathbf{m}$

Exercise 3.44. Implement this alternative transition for `Eval`. What are the other instructions that will need to be changed to allow for this new transition?

Exercise 3.45. What compilation rules would you change to produce optimised code for the boolean operators `&`, `|` and `not`?

Exercise 3.46. Add a `Return` instruction with transition:

(3.48)

	$\mathbf{o}$	`[Return]`	$[\mathbf{a}_0, \ldots, \mathbf{a}_\mathbf{k}]$	$\langle \mathbf{i}, \mathbf{s}\rangle : \mathbf{d}$	$\mathbf{v}$	$\mathbf{h}$	$\mathbf{m}$
$\Longrightarrow$	$\mathbf{o}$	$\mathbf{i}$	$\mathbf{a}_\mathbf{k} : \mathbf{s}$	$\mathbf{d}$	$\mathbf{v}$	$\mathbf{h}$	$\mathbf{m}$

$\mathcal{E}[\![\mathbf{e}]\!]\ \rho$ compiles code that evaluates an expression $\mathbf{e}$ to WHNF in environment ρ, leaving a pointer to the expression on top of the stack.

$$
\begin{array}{rcl}
\mathcal{E}[\![\mathbf{i}]\!]\ \rho & = & [\texttt{Pushint}\ \mathbf{i}] \\
\multicolumn{3}{l}{\mathcal{E}[\![\texttt{let}\ \mathbf{x}_1\texttt{=}\mathbf{e}_1\texttt{;}\ \ldots\texttt{;}\ \mathbf{x_n}\texttt{=}\mathbf{e_n}\ \texttt{in}\ \mathbf{e}]\!]\ \rho} \\
 & = & \mathcal{C}[\![\mathbf{e}_1]\!]\ \rho^{+0} +\!\!+ \ldots +\!\!+ \\
 & & \mathcal{C}[\![\mathbf{e_n}]\!]\ \rho^{+(\mathbf{n}-1)} +\!\!+ \\
 & & \mathcal{E}[\![\mathbf{e}]\!]\ \rho' +\!\!+ [\texttt{Slide}\ \mathbf{n}] \\
 & & \qquad\qquad \text{where } \rho' = \rho^{+\mathbf{n}}[\mathbf{x}_1 \mapsto \mathbf{n}-1, \ldots, \mathbf{x_n} \mapsto 0] \\
\multicolumn{3}{l}{\mathcal{E}[\![\texttt{letrec}\ \mathbf{x}_1\texttt{=}\mathbf{e}_1\texttt{;}\ \ldots\texttt{;}\ \mathbf{x_n}\texttt{=}\mathbf{e_n}\ \texttt{in}\ \mathbf{e}]\!]\ \rho} \\
 & = & [\texttt{Alloc}\ \mathbf{n}] +\!\!+ \\
 & & \mathcal{C}[\![\mathbf{e}_1]\!]\ \rho' +\!\!+ [\texttt{Update}\ \mathbf{n}-1] +\!\!+ \ldots +\!\!+ \\
 & & \mathcal{C}[\![\mathbf{e_n}]\!]\ \rho' +\!\!+ [\texttt{Update}\ 0] +\!\!+ \\
 & & \mathcal{E}[\![\mathbf{e}]\!]\ \rho' +\!\!+ [\texttt{Slide}\ \mathbf{n}] \\
 & & \qquad\qquad \text{where } \rho' = \rho^{+\mathbf{n}}[\mathbf{x}_1 \mapsto \mathbf{n}-1, \ldots, \mathbf{x_n} \mapsto 0] \\
\mathcal{E}[\![\texttt{case}\ \mathbf{e}\ \texttt{of}\ \mathbf{alts}]\!]\ \rho & = & \mathcal{E}[\![\mathbf{e}]\!]\ \rho +\!\!+ [\texttt{Casejump}\ \mathcal{D}_{\mathcal{E}}[\![\mathbf{alts}]\!]\ \rho] \\
\mathcal{E}[\![\texttt{Pack\{}\mathbf{t}\texttt{,}\mathbf{a}\texttt{\}}\ \mathbf{e}_1\ \ldots\ \mathbf{e_a}]\!]\ \rho & = & \mathcal{C}[\![\mathbf{e_a}]\!]\ \rho +\!\!+ \ldots \mathcal{C}[\![\mathbf{e}_1]\!]\ \rho +\!\!+ [\texttt{Pack}\ \mathbf{t}\ \mathbf{a}] \\
\mathcal{E}[\![\mathbf{e}_0\ \texttt{+}\ \mathbf{e}_1]\!]\ \rho & = & \mathcal{B}[\![\mathbf{e}_0\ \texttt{+}\ \mathbf{e}_1]\!]\ \rho +\!\!+ [\texttt{Mkint}] \\
 & & \qquad\qquad \text{And similarly for other arithmetic expressions} \\
\mathcal{E}[\![\mathbf{e}_0\ \texttt{==}\ \mathbf{e}_1]\!]\ \rho & = & \mathcal{B}[\![\mathbf{e}_0\ \texttt{==}\ \mathbf{e}_1]\!]\ \rho +\!\!+ [\texttt{Mkbool}] \\
 & & \qquad\qquad \text{And similarly for other comparison expressions} \\
\mathcal{E}[\![\texttt{negate}\ \mathbf{e}]\!]\ \rho & = & \mathcal{B}[\![\texttt{negate}\ \mathbf{e}]\!]\ \rho +\!\!+ [\texttt{Mkint}] \\
\mathcal{E}[\![\texttt{if}\ \mathbf{e}_0\ \mathbf{e}_1\ \mathbf{e}_2]\!]\ \rho & = & \mathcal{B}[\![\mathbf{e}_0]\!]\ \rho +\!\!+ [\texttt{Cond}\ (\mathcal{E}[\![\mathbf{e}_1]\!]\ \rho)\ (\mathcal{E}[\![\mathbf{e}_2]\!]\ \rho)] \\
\mathcal{E}[\![\mathbf{e}]\!]\ \rho & = & \mathcal{C}[\![\mathbf{e}]\!]\ \rho +\!\!+ [\texttt{Eval}] \qquad\qquad \text{the default case}
\end{array}
$$

$\mathcal{D}_{\mathcal{E}}[\![\mathbf{alts}]\!]\ \rho$ compiles the code for the alternatives in a `case` expression.

$$\mathcal{D}_{\mathcal{E}}[\![\mathbf{alt}_1\ \ldots\ \mathbf{alt_n}]\!]\ \rho \ = \ [\mathcal{A}_{\mathcal{E}}[\![\mathbf{alt}_1]\!]\ \rho,\ \ldots,\ \mathcal{A}_{\mathcal{E}}[\![\mathbf{alt_n}]\!]\ \rho]$$

$\mathcal{A}_{\mathcal{E}}[\![\mathbf{alt}]\!]\ \rho$ compiles the code for an alternative in a `case` expression.

$$
\begin{array}{rcl}
\multicolumn{3}{l}{\mathcal{A}_{\mathcal{E}}[\![\texttt{<t>}\ \mathbf{x}_1 \ldots \mathbf{x_n}\ \texttt{->}\ \mathbf{body}]\!]\ \rho} \\
 & = & \mathbf{t}\ \texttt{->}\ [\texttt{Split}\ \mathbf{n}] +\!\!+ \mathcal{E}[\![\mathbf{body}]\!]\ \rho' +\!\!+ [\texttt{Slide}\ \mathbf{n}] \\
 & & \qquad\qquad \text{where } \rho' = \rho^{+\mathbf{n}}[\mathbf{x}_1 \mapsto 0 \ldots \mathbf{x_n} \mapsto \mathbf{n}-1]
\end{array}
$$

Figure 3.19: The Mark 7 $\mathcal{E}$, $\mathcal{D}_{\mathcal{E}}$ and $\mathcal{A}_{\mathcal{E}}$ compilation schemes

$\mathcal{R}[\![\mathbf{e}]\!]\ \rho\ \mathbf{d}$ generates code which instantiates the expression $\mathbf{e}$ in environment ρ, for a supercombinator of arity $\mathbf{d}$, and then proceeds to unwind the resulting stack.

$$\begin{aligned}
&\mathcal{R}[\![\texttt{let}\ \mathbf{x}_1\texttt{=}\mathbf{e}_1\texttt{;}\ \ldots\texttt{;}\ \mathbf{x_n}\texttt{=}\mathbf{e_n}\ \texttt{in}\ \mathbf{e}]\!]\ \rho\ \mathbf{d} \\
&\quad = \mathcal{C}[\![\mathbf{e}_1]\!]\ \rho^{+0} +\!\!+ \ldots +\!\!+ \\
&\quad\quad \mathcal{C}[\![\mathbf{e_n}]\!]\ \rho^{+(\mathbf{n}-1)} +\!\!+ \\
&\quad\quad \mathcal{R}[\![\mathbf{e}]\!]\ \rho'\ (\mathbf{n}+\mathbf{d}) \\
&\qquad\qquad \text{where } \rho' = \rho^{+\mathbf{n}}[\mathbf{x}_1 \mapsto \mathbf{n}-1, \ldots, \mathbf{x_n} \mapsto 0] \\
&\mathcal{R}[\![\texttt{letrec}\ \mathbf{x}_1\texttt{=}\mathbf{e}_1\texttt{;}\ \ldots\texttt{;}\ \mathbf{x_n}\texttt{=}\mathbf{e_n}\ \texttt{in}\ \mathbf{e}]\!]\ \rho\ \mathbf{d} \\
&\quad = [\texttt{Alloc}\ \mathbf{n}] +\!\!+ \\
&\quad\quad \mathcal{C}[\![\mathbf{e}_1]\!]\ \rho' +\!\!+ [\texttt{Update}\ \mathbf{n}-1] +\!\!+ \ldots +\!\!+ \\
&\quad\quad \mathcal{C}[\![\mathbf{e_n}]\!]\ \rho' +\!\!+ [\texttt{Update}\ 0] +\!\!+ \\
&\quad\quad \mathcal{R}[\![\mathbf{e}]\!]\ \rho'(\mathbf{n}+\mathbf{d}) \\
&\qquad\qquad \text{where } \rho' = \rho^{+\mathbf{n}}[\mathbf{x}_1 \mapsto \mathbf{n}-1, \ldots, \mathbf{x_n} \mapsto 0] \\
&\mathcal{R}[\![\texttt{if}\ \mathbf{e}_0\ \mathbf{e}_1\ \mathbf{e}_2]\!]\ \rho\ \mathbf{d} = \mathcal{B}[\![\mathbf{e}_0]\!]\ \rho +\!\!+ [\texttt{Cond}\ [\mathcal{R}[\![\mathbf{e}_1]\!]\ \rho\ \mathbf{d}, \mathcal{R}[\![\mathbf{e}_2]\!]\ \rho\ \mathbf{d}]] \\
&\mathcal{R}[\![\texttt{case}\ \mathbf{e}\ \texttt{of}\ \mathbf{alts}]\!]\ \rho\ \mathbf{d} = \mathcal{E}[\![\mathbf{e}]\!]\ \rho +\!\!+ [\texttt{Casejump}\ \mathcal{D}_{\mathcal{R}}[\![\mathbf{alts}]\!]\ \rho\ \mathbf{d}] \\
&\mathcal{R}[\![\mathbf{e}]\!]\ \rho\ \mathbf{d} = \mathcal{E}[\![\mathbf{e}]\!]\ \rho +\!\!+ [\texttt{Update}\ \mathbf{d}, \texttt{Pop}\ \mathbf{d}, \texttt{Unwind}] \qquad \text{the default case}
\end{aligned}$$

$\mathcal{D}_{\mathcal{R}}[\![\mathbf{alts}]\!]\ \rho\ \mathbf{d}$ compiles the code for the alternatives in a `case` expression.

$$\mathcal{D}_{\mathcal{R}}[\![\mathbf{alt}_1\ \ldots\ \mathbf{alt_n}]\!]\ \rho\ \mathbf{d} = [\mathcal{A}_{\mathcal{R}}[\![\mathbf{alt}_1]\!]\ \rho\ \mathbf{d},\ \ldots,\ \mathcal{A}_{\mathcal{R}}[\![\mathbf{alt_n}]\!]\ \rho\ \mathbf{d}]$$

$\mathcal{A}_{\mathcal{R}}[\![\mathbf{alt}]\!]\ \rho\ \mathbf{d}$ compiles the code for an alternative in a `case` expression.

$$\begin{aligned}
&\mathcal{A}_{\mathcal{R}}[\![\texttt{<t>}\ \mathbf{x}_1 \ldots \mathbf{x_n}\ \texttt{->}\ \mathbf{body}]\!]\ \rho\ \mathbf{d} \\
&\quad = \mathbf{t}\ \texttt{->}\ [\texttt{Split}\ \mathbf{n}] +\!\!+ \mathcal{R}[\![\mathbf{body}]\!]\ \rho'\ (\mathbf{n}+\mathbf{d}) \\
&\qquad\qquad \text{where } \rho' = \rho^{+\mathbf{n}}[\mathbf{x}_1 \mapsto 0 \ldots \mathbf{x_n} \mapsto \mathbf{n}-1]
\end{aligned}$$

Figure 3.20: The Mark 7 $\mathcal{R}$, $\mathcal{D}_{\mathcal{R}}$, and $\mathcal{A}_{\mathcal{R}}$ compilation schemes

This is used by the $\mathcal{R}$ scheme in place of `Unwind` when the item on top of the stack is known to be in WHNF. Modify `compileR` to generate this new instruction.

Exercise 3.47. Write out the transition for `UpdateInt n`. This instruction performs the same actions as the sequence `[Mkint, Update n]`. Implement this transition and a similar one for `UpdateBool n`. Modify `compileR` to generate these instructions instead of the original sequences.

Why are the new instructions preferred to the original sequences? (Hint: use the statistics from some sample programs.)

3.10 Conclusions

The approach that we have taken in this chapter is a very useful one when designing large pieces of software. First we start with something very simple, and then by a number of gradual changes we produce a very large and complicated piece of software. It would be misleading to claim that this is always possible by a process of small, incremental, changes. In fact the material presented as part of the Mark 1 machine was specifically designed with the Mark 7 machine in mind.

For the moment, however, we have produced a reasonably efficient machine. In the next chapter we will look at the TIM.

Chapter 4

TIM: the three instruction machine

TIM, the Three Instruction Machine, at first appears to be a very different form of reduction machine from those we have seen so far. Nevertheless, it turns out that we can transform a G-machine into a TIM in a series of relatively simple steps. In this chapter we describe these steps, thereby showing how the TIM works, define a complete minimal TIM compiler and evaluator, and then develop a sequence of improvements and optimisations to it.

TIM was invented by Fairbairn and Wray, and their original paper [Fairbairn and Wray 1987] is well worth reading. It describes TIM in a completely different way from the approach taken in this chapter. The material developed in this chapter goes considerably beyond Fairbairn and Wray's work, however, so the level of detail increases in later sections where less well-known ideas are discussed and implemented. Many of the new ideas presented are due to Guy Argo and are presented in his FPCA paper [Argo 1989] and his Ph.D. thesis [Argo 1991].

4.1 Background: How TIM works

Consider the following function definition:

```
f x y = g E1 E2
```

where `E1` and `E2` are arbitrary (and perhaps complex) expressions, and `g` is some other function. Both the template instantiation machine (Chapter 2) and the G-machine (Chapter 3) will perform the following reduction:

```
        @        reduces to      @
       / \                      / \
      @   y                    @   E2
     / \                      / \
    f   x                    g   E1
```

The G-machine will take quite a few (simple) instructions to do this, whereas the template machine does it in one (complicated) step, but the net result is the same.

In this picture, E1 and E2 are the *graphs of* the expressions E1 and E2. For example, if E1 was (x+y)*(x-y), the first argument of g would be a graph of (x+y)*(x-y). This graph has to be laboriously built in the heap (by code generated by the $\mathcal{C}$ compilation scheme). Sadly this might be wasted work, because g might discard its first argument without using it. We would like to find some way of limiting the amount of graph-building done for arguments to functions.

4.1.1 Flattening

Step 1 of our transformation does just this. Suppose we replace the definition of f with the following new one:

```
f x y = g (c1 x y) (c2 x y)
c1 x y = E1
c2 x y = E2
```

We have invented two auxiliary functions, c1 and c2. This definition is plainly equivalent to the old one, but *no matter how large or complicated E1 is, the only work done during the f reduction is to build the graph of (c1 x y).*

Better still, for a G-machine implementation, there is a further benefit which we get automatically. With the first definition, E1 would be compiled by the $\mathcal{C}$ scheme; no advantage can be taken of the optimisations present in the $\mathcal{E}$ scheme when compiling arithmetic expressions. But with the second definition, the expression E1 is now the right-hand side of a supercombinator, so all these optimisations apply. We can evaluate (x+y)*(x-y) much more efficiently in this way.

Of course, E1 and E2 might themselves contain large expressions which will get compiled with the $\mathcal{C}$ scheme (for example, suppose E2 was (h E3 E4)), so we must apply the transformation again to the right-hand sides of c1 and c2. The result is a *flattened* program, so-called because no expression has a nested structure.

4.1.2 Tupling

The next observation is that both c1 and c2 are applied to both x and y, so we have to construct the graphs of (c1 x y) and (c2 x y) before calling g. If c1 and c2 had lots of arguments, rather than just two, the graphs could get quite big. The two graphs are so similar to each other that it is natural to ask whether these argument graphs could share some common part to avoid duplication, and thereby reduce heap allocation. We can express this idea with a second transformation:

```
f x y = let tup = (x,y)
        in g (c1 tup) (c2 tup)
c1 (x,y) = E1
c2 (x,y) = E2
```

The idea is that f first packages up its arguments into a tuple, and then passes this single tuple to c1 and c2. With this definition of f, the f-reduction looks like this:

```
    @       reduces to      @
   / \                     / \
  @   y                   /   @
 / \                     @   / \
f   x                   / \ c2  \
                       g   @     \
                          / \_____\
                         c1        \
                                   -----
                                   |  -|---> x
                                   -----
                                   |  -|---> y
                                   -----
```

4.1.3 Spinelessness

Looking at the previous picture, you can see that *the arguments pointed to by the spine are always of the form* `(c tup)`, for some supercombinator `c` and tuple `tup`. During reduction, we build up a stack of pointers to these arguments. But since they are now all of the same form, we could instead stack the (root of) the arguments themselves! So, after the `f`-reduction, the stack would look like this:

```
|       |       |
|-------------- |
|   c2  |   ----|---\
|---------------|    \    ---------------
|   c1  |   ------------> |     |       | x
|---------------|         |-------------|
                          |     |       | y
                          ---------------
```

Each item on the spine stack is now a pair of a code pointer and a pointer to a tuple. You can think of this pair as an application node, the code defining a function which is being applied to the tuple. On entry to `f`, the (roots of the) arguments `x` and `y` were on the stack, so the tuple of `x` and `y` is actually a tuple of code pointer/tuple pointer pairs.

A code pointer/tuple pointer pair is called a *closure*, and a tuple of such closures is called a *frame*. A pointer to a frame is called a *frame pointer*. Notice that there is no spine in the heap any more; the stack *is* the spine of the expression being evaluated. TIM is a spineless machine.

4.1.4 An example

It is time for an example of how a TIM program might work. Consider the function `compose2`, defined like this:

```
compose2 f g x = f (g x x)
```

The 'flattened' form of `compose2` would be

```
compose2 f g x = f (c1 g x)
c1 g x = g x x
```

When `compose2` is entered, its three arguments will be on top of the stack, like this:

```
    |       |       |
    |---------------|
  x | x-code| x-frm |
    |---------------|
  g | g-code| g-frm |
    |---------------|
  f | f-code| f-frm |
    |---------------|
```

The first thing to do is to form the tuple (frame) of these three arguments in the heap. We can then remove them from the stack. We will keep a pointer to the new frame in a special register, called the *frame pointer*. This is done by the instruction

```
Take 3
```

The state of the machine now looks like this:

```
        |       |       |
        |---------------|
                                  -----------------
Frame ptr ------------------------> f | f-code| f-frm |
                                    |---------------|
                                  g | g-code| g-frm |
                                    |---------------|
                                  x | x-code| x-frm |
                                    -----------------
```

Next, we have to prepare the arguments for `f`. There is only one, namely `(g x x)`, and we want to push a closure for it onto the stack. The frame pointer for the closure is just the current frame pointer register, and so the instruction need only supply a code label:

```
Push (Label "c1")
```

Finally, we want to jump to `f`. Since `f` is an argument to `compose`, not a global supercombinator, `f` is represented by a closure in the current frame. What we must do is fetch the closure, load its frame pointer into the frame pointer register, and its code pointer into the program counter. This is done by the instruction:

```
Enter (Arg 1)           -- f is argument 1
```

After this instruction, the state of the machine is like this:

```
        |       |       |
        |---------------|              ----------------
        |   c1  |   ----|-----------> f | f-code| f-frm |
        |---------------|              |---------------|
                                     g | g-code| g-frm |
  Frame ptr:   f-frm                   |---------------|
  Program ctr: f-code                x | x-code| x-frm |
                                       ----------------
```

That is it! The main body of compose2 consists of just these three instructions:

```
    compose2:   Take 3                  -- 3 arguments
                Push (Label "c1")       -- closure for (g x x)
                Enter (Arg 1)           -- f is argument 1
```

We still need to deal with the label `c1`, though. When the closure for `(g x x)` is needed, it will be entered with the `Enter` instruction, so that the program counter will point to `c1`, and the frame pointer to the original frame containing `f`, `g` and `x`. At this point, all we need do is to prepare the argument for `g`, namely `x`, and enter `g`:

```
        c1:     Push (Arg 3)            -- x is argument 3
                Push (Arg 3)            -- x again
                Enter (Arg 2)           -- g is argument 2
```

The `Push (Arg 3)` instruction fetches a copy of the closure for `x` from the current frame, and pushes it onto the stack. Then the `Enter (Arg 2)` instruction applies `g` to the argument(s) now on the stack[1].

4.1.5 Defining the machine with state transition rules

You can see why it is called the Three Instruction Machine: there are three dominant instructions: `Take`, `Push` and `Enter`. In some ways, it is rather optimistic to claim that it has only three instructions, because `Push` and `Enter` both have several 'addressing modes' and, furthermore, we will need to invent quite a few brand new instructions in due course. Still, it makes a nice name.

As usual, we use state transition rules to express the precise effect of each instruction. First of all we must define the *state* of the machine. It is a quintuple:

(instructions, frame pointer, stack, heap, code store)

or $(\mathbf{i}, \mathbf{f}, \mathbf{s}, \mathbf{h}, \mathbf{c})$ for short. The code store is the only item which has not already been described. It contains a collection of pieces of code, each of which has a label. In practice, the code store contains the compiled supercombinator definitions, each labelled with the name of the

[1]There might be more than just two if the stack was non-empty when the `(g x x)` closure was entered.

supercombinator, though in principle it could also contain other labelled code fragments if that proved useful.

We now develop the transition rules for each of the instructions. `Take` $\mathbf{n}$ forms the top $\mathbf{n}$ elements of the stack into a new frame, and makes the current frame pointer point to it.

$$
(4.1)\qquad
\begin{array}{|lrlrlc|}
\hline
 & \texttt{Take}\ \mathbf{n} : \mathbf{i} & \mathbf{f} & \mathbf{c}_1 : \ldots : \mathbf{c_n} : \mathbf{s} & \mathbf{h} & \mathbf{c} \\
\Longrightarrow & \mathbf{i} & \mathbf{f'} & \mathbf{s} & \mathbf{h}[\mathbf{f'} : \langle \mathbf{c}_1, \ldots, \mathbf{c_n} \rangle] & \mathbf{c} \\
\hline
\end{array}
$$

Now we come to the rules for `Push` and `Enter`. These two instructions have just the same addressing modes (`Arg`, `Label` and so on), and there is a very definite relationship between them, which we dignify with a formal statement:

> The `Push`/`Enter` relationship. *If the instruction `Push` **arg** pushes a closure* $(\mathbf{i}, \mathbf{f})$ *onto the stack, then `Enter` **arg** will load* $\mathbf{i}$ *into the program counter and* $\mathbf{f}$ *into the current frame pointer.*

The instruction `Push (Arg` $\mathbf{n}$`)` fetches the $\mathbf{n}$th closure from the current frame, and pushes it onto the stack.

$$
(4.2)\qquad
\begin{array}{|lrlrlc|}
\hline
 & \texttt{Push (Arg}\ \mathbf{k}\texttt{)} : \mathbf{i} & \mathbf{f} & \mathbf{s} & \mathbf{h}[\mathbf{f} : \langle (\mathbf{i}_1, \mathbf{f}_1), \ldots, (\mathbf{i_k}, \mathbf{f_k}), \ldots, (\mathbf{i_n}, \mathbf{f_n}) \rangle] & \mathbf{c} \\
\Longrightarrow & \mathbf{i} & \mathbf{f} & (\mathbf{i_k}, \mathbf{f_k}) : \mathbf{s} & \mathbf{h} & \mathbf{c} \\
\hline
\end{array}
$$

`Push (Label` $\mathbf{l}$`)` looks up the label $\mathbf{l}$ in the code store, and pushes a closure consisting of this code pointer together with the current frame pointer:

$$
(4.3)\qquad
\begin{array}{|lrlrll|}
\hline
 & \texttt{Push (Label}\ \mathbf{l}\texttt{)} : \mathbf{i} & \mathbf{f} & \mathbf{s} & \mathbf{h} & \mathbf{c}[\mathbf{l} : \mathbf{i'}] \\
\Longrightarrow & \mathbf{i} & \mathbf{f} & (\mathbf{i'}, \mathbf{f}) : \mathbf{s} & \mathbf{h} & \mathbf{c} \\
\hline
\end{array}
$$

In the `compose` example, we had to invent an arbitrary label `c1`. It is a nuisance having to invent these labels, and instead we will simply add a new form for the push instruction, `Push (Code` $\mathbf{i}$`)`, which makes the target code sequence $\mathbf{i}$ part of the instruction itself. Thus, instead of

```
Push (Label "c1")
```

we can write

```
Push (Code [Push (Arg 3), Push (Arg 3), Enter (Arg 2)])
```

Here is the appropriate state transition rule:

$$
(4.4)\qquad
\begin{array}{|lrlrll|}
\hline
 & \texttt{Push (Code}\ \mathbf{i'}\texttt{)} : \mathbf{i} & \mathbf{f} & \mathbf{s} & \mathbf{h} & \mathbf{c} \\
\Longrightarrow & \mathbf{i} & \mathbf{f} & (\mathbf{i'}, \mathbf{f}) : \mathbf{s} & \mathbf{h} & \mathbf{c} \\
\hline
\end{array}
$$

So far we have three 'addressing modes': `Arg`, `Code`, `Label`. We need to add one more, `IntConst`, for integer constants. For example, the call `(f 6)` would compile to the code

```
Push (IntConst 6)
Enter (Label "f")
```

The `Push` instruction always pushes a closure (that is, a code pointer/frame pointer pair) onto the stack, but in the case of integer constants it is not at all obvious what closure it should push. Since we need somewhere to store the integer itself, let us 'steal' the frame pointer slot for that purpose[2]. This decision leads to the following rule, where `intCode` is the (as yet undetermined) code sequence for integer closures:

$$(4.5)\quad \boxed{\begin{array}{lrlrll} & \texttt{Push (IntConst n)} : \mathbf{i} & \mathbf{f} & \mathbf{s} & \mathbf{h} & \mathbf{c} \\ \Longrightarrow & \mathbf{i} & \mathbf{f} & (\texttt{intCode}, \mathbf{n}) : \mathbf{s} & \mathbf{h} & \mathbf{c} \end{array}}$$

What should `intCode` do? For the present our machine will do no arithmetic, so an easy solution is to make `intCode` the empty code sequence:

```
> intCode = []
```

If an integer closure is ever entered, the machine will jump to the empty code sequence, which will halt execution. This will allow us to write programs which return integers, which is enough for Mark 1.

So much for the `Push` instruction. The rules for the `Enter` instruction, one for each addressing mode, follow directly from the `Push`/`Enter` relationship:

$$(4.6)\quad \boxed{\begin{array}{lrllll} & [\texttt{Enter (Label l)}] & \mathbf{f} & \mathbf{s} & \mathbf{h} & \mathbf{c}[\mathbf{l} : \mathbf{i}] \\ \Longrightarrow & \mathbf{i} & \mathbf{f} & \mathbf{s} & \mathbf{h} & \mathbf{c} \end{array}}$$

$$(4.7)\quad \boxed{\begin{array}{lrllll} & [\texttt{Enter (Arg k)}] & \mathbf{f} & \mathbf{s} & \mathbf{h}[\mathbf{f} : \langle(\mathbf{i}_1,\mathbf{f}_1),\ldots,(\mathbf{i}_\mathbf{k},\mathbf{f}_\mathbf{k}),\ldots,(\mathbf{i}_\mathbf{n},\mathbf{f}_\mathbf{n})\rangle] & \mathbf{c} \\ \Longrightarrow & \mathbf{i}_\mathbf{k} & \mathbf{f}_\mathbf{k} & \mathbf{s} & \mathbf{h} & \mathbf{c} \end{array}}$$

$$(4.8)\quad \boxed{\begin{array}{lrllll} & [\texttt{Enter (Code i)}] & \mathbf{f} & \mathbf{s} & \mathbf{h} & \mathbf{c} \\ \Longrightarrow & \mathbf{i} & \mathbf{f} & \mathbf{s} & \mathbf{h} & \mathbf{c} \end{array}}$$

$$(4.9)\quad \boxed{\begin{array}{lrllll} & [\texttt{Enter (IntConst n)}] & \mathbf{f} & \mathbf{s} & \mathbf{h} & \mathbf{c} \\ \Longrightarrow & \texttt{intCode} & \mathbf{n} & \mathbf{s} & \mathbf{h} & \mathbf{c} \end{array}}$$

4.1.6 Compilation

We have now given a precise statement of what each TIM instruction does. It remains to describe how to translate a program into TIM instructions. This we do, as before, using a set of *compilation schemes*. Each supercombinator is compiled with the $\mathcal{SC}$ scheme, which is given in Figure 4.1. The initial environment passed into $\mathcal{SC}$ binds each supercombinator name to a `Label` addressing mode for it. The $\mathcal{SC}$ scheme just produces a `Take` instruction and invokes the

$\mathcal{SC}[\![\mathbf{def}]\!]\ \rho$ is the TIM code for the supercombinator definition **def**, in the environment ρ

$$\mathcal{SC}[\![\mathbf{f}\ \mathbf{x}_1\ \ldots\ \mathbf{x_n}\ \texttt{=}\ \mathbf{e}]\!]\ \rho \quad = \quad \texttt{Take}\ \mathbf{n}\ \texttt{:}\ \mathcal{R}[\![\mathbf{e}]\!]\ \rho[\mathbf{x}_1 \mapsto \texttt{Arg}\ 1, \ldots, \mathbf{x_n} \mapsto \texttt{Arg}\ \mathbf{n}]$$

$\mathcal{R}[\![\mathbf{e}]\!]\ \rho$ is TIM code which applies the value of the expression **e** in environment ρ to the arguments on the stack.

$$\begin{array}{rcll} \mathcal{R}[\![\mathbf{e}_1\ \mathbf{e}_2]\!]\ \rho & = & \texttt{Push}\ (\mathcal{A}[\![\mathbf{e}_2]\!]\ \rho)\ \texttt{:}\ \mathcal{R}[\![\mathbf{e}_1]\!]\ \rho & \\ \mathcal{R}[\![\mathbf{a}]\!]\ \rho & = & \texttt{Enter}\ (\mathcal{A}[\![\mathbf{a}]\!]\ \rho) & \text{where } \mathbf{a} \text{ is an integer, variable,} \\ & & & \text{or supercombinator} \end{array}$$

$\mathcal{A}[\![\mathbf{e}]\!]\ \rho$ is a TIM addressing mode for expression **e** in environment ρ.

$$\begin{array}{rcll} \mathcal{A}[\![\mathbf{x}]\!]\ \rho & = & \rho\,\mathbf{x} & \text{where } \mathbf{x} \text{ is bound by } \rho \\ \mathcal{A}[\![\mathbf{n}]\!]\ \rho & = & \texttt{IntConst}\ \mathbf{n} & \text{where } \mathbf{n} \text{ is an integer} \\ \mathcal{A}[\![\mathbf{e}]\!]\ \rho & = & \texttt{Code}\ (\mathcal{R}[\![\mathbf{e}]\!]\ \rho) & \text{otherwise} \end{array}$$

Figure 4.1: The $\mathcal{SC}$, $\mathcal{R}$ and $\mathcal{A}$ compilation schemes

$\mathcal{R}$ scheme, passing it an environment augmented by bindings which say what addressing mode to use for each argument.

The $\mathcal{R}$ scheme (Figure 4.1) simply pushes arguments onto the stack until it finds a variable or supercombinator, which it enters. It uses the $\mathcal{A}$ scheme to generate the correct addressing mode. Notice the way that the flattening process described in Section 4.1.1 is carried out 'on the fly' by these rules.

For the present, we omit arithmetic, data structures, case analysis and `let(rec)` expressions. They will all be added later.

4.1.7 Updating

So far there has been no mention of updating. That is because, now that the spine has vanished, there are no spine nodes to update! Indeed, the machine as so far described is a tree-reduction machine. Shared arguments may be evaluated repeatedly. Doing updates properly is the Achilles' heel of spineless implementations. It is utterly necessary, because otherwise an unbounded amount of work could be duplicated, yet it adds complexity which loses some of the elegance and speed (duplication aside) of the non-updating version.

We will return to updating later in Section 4.5, but meanwhile it is enough to implement the non-updating version.

[2] We are making the implicit assumption that an integer is no larger than a frame pointer, which is usually true in practice.

4.2 Mark 1: A minimal TIM

In this section we will develop a minimal, but complete, TIM implementation, without arithmetic, data structures or updates. These will be added in subsequent sections.

4.2.1 Overall structure

The structure is much the same as for the template instantiation interpreter. The `run` function is the composition of four functions, `parse`, `compile`, `eval` and `showResults`, just as before. The type of `parse` is given in Chapter 1; the types for the other three are given below:

```
> run         :: [char] -> [char]
> compile     :: coreProgram -> timState
> eval        :: timState -> [timState]
> showResults :: [timState] -> [char]
>
> run = showResults . eval . compile . parse
```

It is often convenient to see all the intermediate states, so we also provide `fullRun`, which uses `showFullResults` to show each state:

```
> fullRun :: [char] -> [char]
> fullRun = showFullResults . eval . compile . parse
```

We need to import the language module:

```
> %include "language"
```

4.2.2 Data type definitions

The data type for TIM instructions corresponds directly to the instructions introduced so far.

```
> instruction ::= Take num
>               | Enter timAMode
>               | Push timAMode
```

The type of addressing modes, `timAMode`, is separated out as a distinct data type to stress the relationship between `Push` and `Enter`.

```
> timAMode ::= Arg num
>            | Label [char]
>            | Code [instruction]
>            | IntConst num
```

The state of the TIM machine is given by the following definition:

```
> timState == ([instruction],        || The current instruction stream
>              framePtr,             || Address of current frame
>              timStack,             || Stack of arguments
>              timValueStack,        || Value stack (not used yet)
>              timDump,              || Dump (not used yet)
>              timHeap,              || Heap of frames
>              codeStore,            || Labelled blocks of code
>              timStats)             || Statistics
```

The value stack and dump are only required later on in this chapter, but it is more convenient to add placeholders for them right away.

We consider the representation for each of these components in turn.

- The *current instruction stream* is represented by a list of instructions. In a real machine this would be the program counter together with the program memory.
- The *frame pointer* is usually the address of a frame in the heap, but there are two other possibilities: it might be used to hold an integer value, or it might be uninitialised. The machine always 'knows' which of these three possibilities to expect, but it is convenient in our implementation to distinguish them by using an algebraic data type for `framePtr`:

  ```
  > framePtr ::= FrameAddr addr           || The address of a frame
  >              | FrameInt num           || An integer value
  >              | FrameNull              || Uninitialised
  ```

  If we do not do this, Miranda will (legitimately) complain of a type error when we try to use an address as a number. Furthermore, having a constructor for the uninitialised state `FrameNull` means that our interpreter will discover if we ever mistakenly try to use an uninitialised value as a valid address.
- The *stack* contains *closures*, each of which is a pair containing a code pointer and a frame pointer. We represent the stack as a list.

  ```
  > timStack == [closure]
  > closure == ([instruction], framePtr)
  ```

- The *value stack* and *dump* are not used at all to begin with, so we represent each of them with a dummy algebraic data type which has just one nullary constructor. Later we will replace these definitions with more interesting ones.

  ```
  > timValueStack ::= DummyTimValueStack
  > timDump ::= DummyTimDump
  ```

- The *heap* contains *frames*, each of which is a tuple of closures. The data type of frames is important enough to merit an abstract data type of its own.

```
> timHeap == heap frame
>
> abstype frame
> with fAlloc   :: timHeap -> [closure] -> (timHeap, framePtr)
>      fGet     :: timHeap -> framePtr -> num -> closure
>      fUpdate  :: timHeap -> framePtr -> num -> closure -> timHeap
>      fList    :: frame -> [closure]           || Used when printing
```

These operations allow frames to be built, and components to be extracted and updated. The first element of the list given to `fAlloc` is numbered `1` for the purposes of `fGet` and `fUpdate`. Here is a simple implementation based on lists.

```
> frame == [closure]
>
> fAlloc heap xs = (heap', FrameAddr addr)
>                  where
>                  (heap', addr) = hAlloc heap xs
>
> fGet heap (FrameAddr addr) n = f ! (n-1)      || Miranda's ! operator
>                                               || uses zero indexing
>                                where
>                                f = hLookup heap addr
>
> fUpdate heap (FrameAddr addr) n closure
>  = hUpdate heap addr new_frame
>    where
>    frame = hLookup heap addr
>    new_frame = take (n-1) frame ++ [closure] ++ drop n frame
>
> fList f = f
```

- For each label, the *code store* gives the corresponding compiled code:

  ```
  > codeStore == assoc name [instruction]
  ```

  We take the opportunity to provide a lookup function for labels, which generates an error message if it fails:

  ```
  > codeLookup :: codeStore -> name -> [instruction]
  > codeLookup cstore l
  >  = aLookup cstore l (error ("Attempt to jump to unknown label "
  >                             ++ show l))
  ```

- As usual, we make the *statistics* into an abstract data type which we can add to easily:

  ```
  > abstype timStats
  > with  statInitial  :: timStats
  >       statIncSteps :: timStats -> timStats
  >       statGetSteps :: timStats -> num
  ```

The first implementation, which counts only the number of steps, is rather simple:

```
> timStats == num                || The number of steps
> statInitial = 0
> statIncSteps s = s+1
> statGetSteps s = s
```

Finally, we need the code for heaps and stacks:

```
> %include "utils"
```

4.2.3 Compiling a program

`compile` works very much like the template instantiation compiler, creating an initial machine state from the program it is given. The main difference lies in the compilation function `compileSC` which is applied to each supercombinator.

```
> compile program
>     = ([Enter (Label "main")],     || Initial instructions
>        FrameNull,                  || Null frame pointer
>        initialArgStack,            || Argument stack
>        initialValueStack,          || Value stack
>        initialDump,                || Dump
>        hInitial,                   || Empty heap
>        compiled_code,              || Compiled code for supercombinators
>        statInitial)                || Initial statistics
>        where
>        sc_defs          = preludeDefs ++ program
>        compiled_sc_defs = map (compileSC initial_env) sc_defs
>        compiled_code    = compiled_sc_defs ++ compiledPrimitives
>        initial_env = [(name, Label name) | (name, args, body) <- sc_defs]
>                      ++ [(name, Label name) | (name, code) <- compiledPrimitives]
```

For the moment, the argument stack is initialised to be empty.

```
> initialArgStack = []
```

For now the value stack and dump are initialised to their dummy values. Later we will change these definitions.

```
> initialValueStack = DummyTimValueStack
> initialDump = DummyTimDump
```

The compiled supercombinators, `compiled_sc_defs`, is obtained by compiling each of the supercombinators in the program, using `compileSC`. The initial environment passed to `compileSC` gives a suitable addressing mode for each supercombinator. The code store, `compiled_code`, is obtained by combining `compiled_sc_defs` with `compiledPrimitives`. The latter is intended to contain compiled code for built-in primitives, but it is empty for the present:

```
> compiledPrimitives = []
```

Unlike the template machine and the G-machine, the initial heap is empty. The reason for a non-empty initial heap in those cases was to retain sharing for CAFs (that is, supercombinators with no arguments – Section 2.1.6). In this initial version of the TIM machine, the compiled TIM code for a CAF will be executed each time it is called, so the work of evaluating the CAF is not shared. We will address this problem much later, in Section 4.7.

The heart of the compiler is a direct translation of the compilation schemes $\mathcal{SC}$, $\mathcal{R}$ and $\mathcal{A}$ into the functions `compileSC`, `compileR` and `compileA` respectively. The environment, ρ, is represented by an association list binding names to addressing modes. The G-machine compiler used a mapping from names to stack offsets, but the extra flexibility of using addressing modes turns out to be rather useful.

```
> timCompilerEnv == [(name, timAMode)]
```

Now we are ready to define `compileSC`:

```
> compileSC :: timCompilerEnv -> coreScDefn -> (name, [instruction])
> compileSC env (name, args, body)
>  = (name, Take (#args) : instructions)
>     where
>     instructions = compileR body new_env
>     new_env = (zip2 args (map Arg [1..])) ++ env
```

`compileR` takes an expression and an environment, and delivers a list of instructions:

```
> compileR :: coreExpr -> timCompilerEnv -> [instruction]
> compileR (EAp e1 e2) env = Push (compileA e2 env) : compileR e1 env
> compileR (EVar v)    env = [Enter (compileA (EVar v) env)]
> compileR (ENum n)    env = [Enter (compileA (ENum n) env)]
> compileR e           env = error "compileR: can't do this yet"
```

```
> compileA :: coreExpr -> timCompilerEnv -> timAMode
> compileA (EVar v) env = aLookup env v (error ("Unknown variable " ++ v))
> compileA (ENum n) env = IntConst n
> compileA e        env = Code (compileR e env)
```

4.2.4 The evaluator

Next we need to define how the evaluator actually works. The definition of `eval` is exactly as for the template instantiation machine:

```
> eval state
> = state : rest_states  where
```

```
>                         rest_states = [],               timFinal state
>                                     = eval next_state,  otherwise
>                         next_state  = doAdmin (step state)
>
> doAdmin state = applyToStats statIncSteps state
```

The timFinal function says when a state is a final state. We could invent a Stop instruction, but it is just as easy to say that we have finished when the code sequence is empty:

```
> timFinal ([], frame, stack, vstack, dump, heap, cstore, stats) = True
> timFinal state                                                 = False
```

The applyToStats function just applies a function to the statistics component of the state:

```
> applyToStats stats_fun (instr, frame, stack, vstack,
>                          dump, heap, cstore, stats)
>  = (instr, frame, stack, vstack, dump, heap, cstore, stats_fun stats)
```

Taking a step

step does the case analysis which takes a single instruction and executes it. The Take equation is a straightforward transliteration of the corresponding state transition rule (4.1):

```
> step ((Take n:instr), fptr, stack, vstack, dump, heap, cstore,stats)
>  = (instr, fptr', drop n stack, vstack, dump, heap', cstore, stats), #stack >= n
>  = error "Too few args for Take instruction",                            otherwise
>    where (heap', fptr') = fAlloc heap (take n stack)
```

The equations for Enter and Push take advantage of the Push/Enter relationship by using a common function amToClosure which converts a timAMode to a closure:

```
> step ([Enter am], fptr, stack, vstack, dump, heap, cstore, stats)
>  = (instr', fptr', stack, vstack, dump, heap, cstore, stats)
>    where (instr',fptr') = amToClosure am fptr heap cstore


> step ((Push am:instr), fptr, stack, vstack, dump, heap, cstore, stats)
>  = (instr, fptr, amToClosure am fptr heap cstore : stack,
>     vstack, dump, heap, cstore, stats)
```

amToClosure delivers the closure addressed by the addressing mode which is its first argument:

```
> amToClosure :: timAMode -> framePtr -> timHeap -> codeStore -> closure
> amToClosure (Arg n)      fptr heap cstore = fGet heap fptr n
> amToClosure (Code il)    fptr heap cstore = (il, fptr)
> amToClosure (Label l)    fptr heap cstore = (codeLookup cstore l, fptr)
> amToClosure (IntConst n) fptr heap cstore = (intCode, FrameInt n)
```

4.2.5 Printing the results

As with the template instantiation version we need a rather boring collection of functions to print the results in a sensible way. It is often useful to print out the supercombinator definitions, so `showResults` begins by doing so, using the definitions in the first state:

```
> showFullResults states
>  = iDisplay (iConcat [
>        iStr "Supercombinator definitions", iNewline, iNewline,
>        showSCDefns first_state, iNewline, iNewline,
>        iStr "State transitions", iNewline,
>        iLayn (map showState states), iNewline, iNewline,
>        showStats (last states)
>    ])
>    where
>    (first_state:rest_states) = states
```

`showResults` just shows the last state and some statistics:

```
> showResults states
>  = iDisplay (iConcat [
>     showState last_state, iNewline, iNewline, showStats last_state
>    ])
>    where last_state = last states
```

The rest of the functions are straightforward. `showSCDefns` displays the code for each supercombinator.

```
> showSCDefns :: timState -> iseq
> showSCDefns (instr, fptr, stack, vstack, dump, heap, cstore, stats)
>  = iInterleave iNewline (map showSC cstore)
```

```
> showSC :: (name, [instruction]) -> iseq
> showSC (name, il)
>  = iConcat [
>        iStr "Code for ", iStr name, iStr ":", iNewline,
>        iStr "   ", showInstructions Full il, iNewline, iNewline
>    ]
```

`showState` displays a TIM machine state.

```
> showState :: timState -> iseq
> showState (instr, fptr, stack, vstack, dump, heap, cstore, stats)
>  = iConcat [
>     iStr "Code:  ", showInstructions Terse instr, iNewline,
>     showFrame heap fptr,
```

```
>     showStack stack,
>     showValueStack vstack,
>     showDump dump,
>     iNewline
>    ]
```

showFrame shows the frame component of a state, using showClosure to display each of the closures inside it.

```
> showFrame :: timHeap -> framePtr -> iseq
> showFrame heap FrameNull = iStr "Null frame ptr" $iAppend iNewline
> showFrame heap (FrameAddr addr)
>  = iConcat [
>        iStr "Frame: <",
>        iIndent (iInterleave iNewline
>                             (map showClosure (fList (hLookup heap addr)))),
>        iStr ">", iNewline
>    ]
> showFrame heap (FrameInt n)
>  = iConcat [ iStr "Frame ptr (int): ", iNum n, iNewline ]
```

showStack displays the argument stack, using showClosure to display each closure.

```
> showStack :: timStack -> iseq
> showStack stack
>  = iConcat [   iStr "Arg stack: [",
>                iIndent (iInterleave iNewline (map showClosure stack)),
>                iStr "]", iNewline
>    ]
```

For the present, showValueStack and showDump, which display the value stack and dump, are stubs for now, because we are not using these components of the state.

```
> showValueStack :: timValueStack -> iseq
> showValueStack vstack = iNil
```

```
> showDump :: timDump -> iseq
> showDump dump = iNil
```

showClosure displays a closure, using showFramePtr to display the frame pointer.

```
> showClosure :: closure -> iseq
> showClosure (i,f)
>  = iConcat [   iStr "(",  showInstructions Terse i,  iStr ", ",
>                showFramePtr f,  iStr ")"
>    ]
```

```
> showFramePtr :: framePtr -> iseq
> showFramePtr FrameNull = iStr "null"
> showFramePtr (FrameAddr a) = iStr (show a)
> showFramePtr (FrameInt n) = iStr "int " $iAppend iNum n
```

showStats is responsible for printing out accumulated statistics:

```
> showStats :: timState -> iseq
> showStats (instr, fptr, stack, vstack, dump, heap, code, stats)
>  = iConcat [ iStr "Steps taken = ", iNum (statGetSteps stats), iNewline,
>              iStr "No of frames allocated = ", iNum (hSize heap),
>              iNewline
>    ]
```

Printing instructions

We are going to need to print instructions and instruction sequences. If a sequence of instructions is printed as one long line, it is rather hard to read, so it is worth writing some code to pretty-print them.

In fact we want to be able to print either the entire code for an instruction sequence (for example when printing a supercombinator definition), or just some abbreviated form of it. An example of the latter occurs when printing the contents of the stack; it can be helpful to see some part of the code in each closure, but we do not want to see it all! Accordingly, we give an extra argument, `d`, to each function to tell it how fully to print. The value of this argument is either `Full`, `Terse` or `None`.

```
> howMuchToPrint ::= Full | Terse | None
```

`showInstructions` turns a list of instructions into an `iseq`. When `d` is `None`, only an ellipsis is printed. If `d` is `Terse`, the instructions are printed all on one line, and nested instructions are printed with `d` as `None`. If `d` is `Full`, the instructions are laid out one per line, and printed in full.

```
> showInstructions :: howMuchToPrint -> [instruction] -> iseq
> showInstructions None il = iStr "{..}"
> showInstructions Terse il
>  = iConcat [iStr "{", iIndent (iInterleave (iStr ", ") body), iStr "}"]
>    where
>       instrs = map (showInstruction None) il
>       body = instrs,                  #il <= nTerse
>            = (take nTerse instrs)
>              ++ [iStr ".."],          otherwise
> showInstructions Full il
>  = iConcat [iStr "{ ", iIndent (iInterleave sep instrs), iStr " }"]
>    where
>    sep = iStr "," $iAppend iNewline
>    instrs = map (showInstruction Full) il
```

`showInstruction` turns a single instruction into an `iseq`.

```
> showInstruction d (Take m)  = (iStr "Take ")  $iAppend (iNum m)
> showInstruction d (Enter x) = (iStr "Enter ") $iAppend (showArg d x)
> showInstruction d (Push x)  = (iStr "Push ")  $iAppend (showArg d x)
```

```
> showArg d (Arg m)      = (iStr "Arg ")   $iAppend (iNum m)
> showArg d (Code il)    = (iStr "Code ")  $iAppend (showInstructions d il)
> showArg d (Label s)    = (iStr "Label ") $iAppend (iStr s)
> showArg d (IntConst n) = (iStr "IntConst ") $iAppend (iNum n)
```

`nTerse` says how many instructions of a sequence should be printed in terse form.

```
> nTerse = 3
```

Exercise 4.1. Run the machine using the following definition of `main`:

```
main = S K K 4
```

Since `S K K` is the identity function, `main` should reduce to `4`, which halts the machine. Experiment with making it a little more elaborate; for example

```
id = S K K ;
id1 = id id ;
main = id1 4
```

Exercise 4.2. Add more performance instrumentation. For example:

- Measure execution time, counting one time unit for each instruction except `Take`, for which you should count as many time units as the frame has elements.
- Measure the the heap usage, printing the total amount of heap allocated in a run. Take account of the size of the frames, so that you can compare your results directly with those from the template instantiation version.
- Measure the maximum stack depth.

Exercise 4.3. If $\mathbf{n} = 0$, then `Take` $\mathbf{n}$ does nothing useful. Adapt the definition of `compileSC` to spot this optimisation by omitting the `Take` instruction altogether for CAFs.

4.2.6 Garbage collection†

Like any heap-based system, TIM requires a garbage collector, but it also requires one with a little added sophistication. As usual, the garbage collector finds all the live data by starting from the machine state; that is, from the stack and the frame pointer. Each closure on the stack points to a frame, which must clearly be retained. But that frame in turn contains pointers to further frames, and so on. The question arises: *given a particular frame, which frame pointers within it should be recursively followed?*

The safe answer is 'follow all of them', but this risks retaining far more data than required. For example, the closure for `(g x x)` in the `compose2` example of Section 4.1.4 has a pointer to a frame containing `f`, `g` and `x`, but it only requires the closures for `g` and `x`. A naive garbage collector might follow the frame pointer from `f`'s closure as well, thus retaining data unnecessarily. This unwanted retention is called a *space leak*, and can cause garbage collection to occur much more frequently than would otherwise be the case.

However, this particular space leak is straightforward, if rather tedious, to eliminate. Each closure consists of a code pointer paired with a frame pointer. The code 'knows' which frame elements it is going to need, and this information can be recorded with the code, for the garbage collector to examine. For example, what we have been calling a 'code pointer' could actually point to a pair, consisting of a list of slot numbers used by the code, and the code itself. (In a real implementation the list might be encoded as a bit-mask.) How can the list of useful slots be derived? It is simple: just find the free variables of the expression being compiled, and use the environment to map them into slot numbers.

4.3 Mark 2: Adding arithmetic

In this section we will add arithmetic to our machine.

4.3.1 Overview: how arithmetic works

The original Fairbairn and Wray TIM machine had a rather devious scheme for doing arithmetic. Their main motivation was to keep the machine *minimal*, but their approach is quite hard to understand and requires considerable massaging to give an efficient implementation.

Instead, we will modify the TIM in a way exactly analogous to the V-stack of the G-machine (Section 3.9). We modify the state by introducing a *value stack*, which is a stack of (evaluated, unboxed) integers. We extend the instruction set with a family of instructions `Op` **op** which perform the arithmetic operation **op** on the top elements of the value stack, leaving the result on top of the value stack. For example, the `Op Sub` instruction removes the top two elements of the value stack, subtracts them and pushes the result onto the value stack:

$$(4.10)\quad \begin{array}{|rrrrrrr|} & \texttt{Op Sub} : \mathbf{i} & \mathbf{f} & \mathbf{s} & \mathbf{n}_1 : \mathbf{n}_2 : \mathbf{v} & \mathbf{h} & \mathbf{c} \\ \Longrightarrow & \mathbf{i} & \mathbf{f} & \mathbf{s} & (\mathbf{n}_1 - \mathbf{n}_2) : \mathbf{v} & \mathbf{h} & \mathbf{c} \end{array}$$

It is easy to define a complete family of arithmetic instructions, `Op Add`, `Op Sub`, `Op Mult`, `Op Div`, `Op Neg` and so on, in this way.

Now consider the following function `sub`:

```
sub a b = a - b
```

What code should we generate for `sub`? It has to take the following steps:

1. The usual `Take 2` to form its arguments into a frame.
2. Evaluate `b`, putting its value on the value stack.
3. Evaluate `a`, doing likewise.

4. Subtract the value of `b` from the value of `a`, using the `Op Sub` instruction, which leaves its result on top of the value stack.

5. 'Return' to the 'caller'.

We will consider the evaluation of `a` and `b` first. They are represented by closures, held in the current frame, and the only thing we can do to a closure is to enter it. So presumably to evaluate `a` we must enter the closure for `a`, but what does it mean to enter an integer-valued closure? So far we have only entered *functions*, and integers are not functions. Here is the key idea:

> Integer invariant: *when an integer-valued closure is entered, it computes the value of the integer, pushes it onto the value stack, and enters the top closure on the argument stack.*

The closure on top of the argument stack is called the *continuation*, because it says what to do next, once the evaluation of the integer is complete. The continuation consists of an instruction sequence, saying what to do when evaluation of the integer is complete, and the current frame pointer (in case it was disturbed by the evaluation of the integer). In other words, the continuation is a perfectly ordinary closure.

So the code for `sub` looks like this:

```
sub:    Take 2
        Push (Label L1)         -- Push the continuation
        Enter (Arg 2)           -- Evaluate b

L1:     Push (Label L2)         -- Push another continuation
        Enter (Arg 1)           -- Evaluate a

L2:     Op Sub                  -- Compute a-b on value stack
        Return
```

What should the `Return` instruction do? Since the value returned by `sub` is an integer, and after the `Op Sub` instruction this integer is on top of the value stack, all `Return` has to do is to pop the closure on top of the argument stack and enter it:

$$
(4.11)\quad \boxed{\begin{array}{lrlrcccc} & [\mathtt{Return}] & \mathbf{f} & (\mathbf{i}',\mathbf{f}') : \mathbf{s} & \mathbf{v} & \mathbf{h} & \mathbf{c} \\ \Longrightarrow & \mathbf{i}' & \mathbf{f}' & \mathbf{s} & \mathbf{v} & \mathbf{h} & \mathbf{c} \end{array}}
$$

We have used labels to write the code for `sub`. This is not the only way to do it; an alternative is to use the `Push Code` instruction, which avoids the tiresome necessity of inventing new labels. In this style the code for `sub` becomes:

```
sub:    Take 2
        Push (Code [    Push (Code [Op Sub, Return]),
                        Enter (Arg 1)
             ])
        Enter (Arg 2)
```

Written like this, it is less easy to see what is going on than by using labels, so we will continue to use labels in the exposition where it makes code fragments easier to understand, but we will use the `Push Code` version in the compiler.

Now we must return to the question of integer constants. Consider the expression `(sub 4 2)`. It will compile to the code

```
Push (IntConst 2)
Push (IntConst 4)
Enter (Label "sub")
```

The code for `sub` will soon enter the closure `(IntConst 2)`, which will place the integer `2` in the frame pointer and jump to `intCode`. Currently, `intCode` is the empty code sequence (so that the machine stops if we ever enter an integer), but we need to change that. What should `intCode` now do? The answer is given by the integer invariant: it must push the integer onto the value stack and return, thus:

```
> intCode = [PushV FramePtr, Return]
```

`PushV FramePtr` is a new instruction which pushes the number currently masquerading as the frame pointer onto the top of the value stack:

(4.12)
$$\begin{array}{|lrccccc|}\hline & \texttt{PushV FramePtr} : \mathbf{i} & \mathbf{n} & \mathbf{s} & & \mathbf{v} & \mathbf{h} \;\; \mathbf{c} \\ \Longrightarrow & \mathbf{i} & \mathbf{n} & \mathbf{s} & \mathbf{n} : & \mathbf{v} & \mathbf{h} \;\; \mathbf{c} \\ \hline\end{array}$$

4.3.2 Adding simple arithmetic to the implementation

Now we are ready to modify our implementation. We keep the modifications to a minimum by adding code for each of the arithmetic functions to `compiledPrimitives`. Recall that when we write (for example) `p-q` in a program, the parser converts it to

```
EAp (EAp (EVar "-") (EVar "p")) (EVar "q")
```

All we need do is to work out some suitable code for the primitive `-`, and add this code to the code store. The compiler can then treat `-` in the same way as any other supercombinator. Finally, the code for `-` that we want is exactly that which we developed in the previous section for `sub`, and similar code is easy to write for other arithmetic operations.

So the steps required are as follows:

- Add the following type definition and initialisation for the value stack:

  ```
  > timValueStack == [num]
  > initialValueStack = []
  ```

- Add the new instructions `PushV`, `Return` and `Op` to the `instruction` data type. We take the opportunity to add one further instruction, `Cond`, which has not yet been discussed but is the subject of a later exercise. TIM is no longer a three instruction machine!

```
> instruction ::= Take num
>               | Push timAMode
>               | PushV valueAMode
>               | Enter timAMode
>               | Return
>               | Op op
>               | Cond [instruction] [instruction]
>
> op ::= Add  | Sub | Mult | Div | Neg
>      | Gr | GrEq | Lt | LtEq | Eq | NotEq
```

So far the argument of a `PushV` instruction can only be `FramePtr`, but we will shortly add a second form which allows us to push literal constants onto the value stack. So it is worth declaring an algebraic data type for `valueAMode`:

```
> valueAMode ::= FramePtr
>              | IntVConst num
```

The `showInstruction` function must be altered to deal with this additional structure.

- Modify the `step` function to implement the extra instructions. This is just a question of translating the state transition rules into Miranda.
- Add to `compiledPrimitives` suitable definitions for `+`, `-` and so on.
- Now that `intCode` is no longer empty, we must initialise the stack to have a suitable continuation (return address) for `main` to return to. The way to do this is to make `compile` initialise the stack with the closure `([],FrameNull)`, by redefining `initialArgStack`:

```
> initialArgStack = [([], FrameNull)]
```

This continuation has an empty code sequence, so the machine will now halt with the result on top of the value stack.

Exercise 4.4. Implement these changes on your prototype. Try it out on some simple examples; for example

```
four = 2 * 2
main = four + four
```

Exercise 4.5. We still cannot execute 'interesting' programs, because we do not yet have a conditional, and without a conditional we cannot use recursion. A simple solution is to add a new instruction `Cond i1 i2`, which removes a value from the top of the value stack, checks whether it was zero and if so continues with instruction sequence `i1`, otherwise continues with `i2`. Here are its state transition rules:

(4.13)

	[Cond $\mathbf{i}_1$ $\mathbf{i}_2$]	$\mathbf{f}$	$\mathbf{s}$	$0 : \mathbf{v}$	$\mathbf{h}$	$\mathbf{c}$
$\Longrightarrow$	$\mathbf{i}_1$	$\mathbf{f}$	$\mathbf{s}$	$\mathbf{v}$	$\mathbf{h}$	$\mathbf{c}$
	[Cond $\mathbf{i}_1$ $\mathbf{i}_2$]	$\mathbf{f}$	$\mathbf{s}$	$\mathbf{n} : \mathbf{v}$	$\mathbf{h}$	$\mathbf{c}$
$\Longrightarrow$	$\mathbf{i}_2$	$\mathbf{f}$	$\mathbf{s}$	$\mathbf{v}$	$\mathbf{h}$	$\mathbf{c}$
	where $\mathbf{n} \neq 0$					

The first rule matches if zero is on top of the value stack; otherwise the second rule applies.

You also need to add a primitive `if`, which behaves as follows:

```
if 0 t f = t
if n t f = f
```

You need to work out the TIM code for `if`, using the `Cond` instruction, and add it to `compiledPrimitives`. Finally, you can test your improved system with the factorial function:

```
factorial n = if n 1 (n * factorial (n-1))
main = factorial 3
```

4.3.3 Compilation schemes for arithmetic

Just as with the G-machine, we can do a much better job of compiling for our machine than we are doing at present. Consider a function such as

```
f x y z = (x+y) * z
```

As things stand, this will get parsed to

```
f x y z = * (+ x y) z
```

and code for `f` will get compiled which will call the standard functions `*` and `+`. But we could do much better than this! Instead of building a closure for `(+ x y)` and passing it to `*`, for example, we can just do the operations in-line, using the following steps:

1. evaluate `x`
2. evaluate `y`
3. add them
4. evaluate `z`
5. multiply
6. return

No closures need be built and no jumps need occur (except those needed to evaluate `x`, `y` and `z`).

To express this improvement, we introduce a new compilation scheme to deal with expressions whose value is an integer, the $\mathcal{B}$ scheme. It is defined like this: for any expression **e** whose value is an integer, and for any code sequence **cont**,

> ($\mathcal{B}[\![\mathbf{e}]\!]\ \rho$ **cont**) *is a code sequence which, when executed with a current frame laid out as described by* ρ*, will push the value of the expression* **e** *onto the value stack, and then execute the code sequence* **cont***.*

$\mathcal{R}[\![\mathbf{e}]\!]\ \rho$ is TIM code which applies the value of the expression **e** in environment ρ to the arguments on the stack.		
$\mathcal{R}[\![\mathbf{e}]\!]\ \rho = \mathcal{B}[\![\mathbf{e}]\!]\ \rho$ `[Return]`	where **e** is an arithmetic expression, such as $\mathbf{e}_1 + \mathbf{e}_2$, or a number	
$\mathcal{R}[\![\mathbf{e}_1\ \mathbf{e}_2]\!]\ \rho =$ `Push` $(\mathcal{A}[\![\mathbf{e}_2]\!]\ \rho)$ `:` $\mathcal{R}[\![\mathbf{e}_1]\!]\ \rho$		
$\mathcal{R}[\![\mathbf{a}]\!]\ \rho =$ `Enter` $(\mathcal{A}[\![\mathbf{a}]\!]\ \rho)$	where **a** is a variable, or supercombinator	

$\mathcal{A}[\![\mathbf{e}]\!]\ \rho$ is a TIM addressing mode for expression **e** in environment ρ.

$\mathcal{A}[\![\mathbf{x}]\!]\ \rho$	$= \rho\,\mathbf{x}$	where **x** is bound by ρ
$\mathcal{A}[\![\mathbf{n}]\!]\ \rho$	$=$ `IntConst` **n**	where **n** is an integer constant
$\mathcal{A}[\![\mathbf{e}]\!]\ \rho$	$=$ `Code` $(\mathcal{R}[\![\mathbf{e}]\!]\ \rho)$	otherwise

$\mathcal{B}[\![\mathbf{e}]\!]\ \rho$ **cont** is TIM code which evaluates **e** in environment ρ, and puts its value, which should be an integer, on top of the value stack, and then continues with the code sequence **cont**.

$\mathcal{B}[\![\mathbf{e}_1$ `+` $\mathbf{e}_2]\!]\ \rho$ **cont** $= \mathcal{B}[\![\mathbf{e}_2]\!]\ \rho\ (\mathcal{B}[\![\mathbf{e}_1]\!]\ \rho$ (`Op Add :` **cont**))

... and similar rules for other arithmetic primitives

$\mathcal{B}[\![\mathbf{n}]\!]\ \rho$ **cont**	$=$ `PushV (IntVConst` **n**`) :` **cont**	where **n** is a number
$\mathcal{B}[\![\mathbf{e}]\!]\ \rho$ **cont**	$=$ `Push (Code` **cont**`) :` $\mathcal{R}[\![\mathbf{e}]\!]\ \rho$	otherwise

Figure 4.2: Revised compilation schemes for arithmetic

The compilation scheme uses a *continuation-passing style*, in which the **cont** argument says what to do after the value has been computed. Figure 4.2 gives the $\mathcal{B}$ compilation scheme, together with the revised $\mathcal{R}$ and $\mathcal{A}$ schemes. When $\mathcal{R}$ finds an expression which is an arithmetic expression it calls $\mathcal{B}$ to compile it. $\mathcal{B}$ has special cases for constants and applications of arithmetic operators, which avoid explicitly pushing the continuation. If it encounters an expression which it cannot handle specially, it just pushes the continuation and calls $\mathcal{R}$.

There is one new instruction required, which is used when $\mathcal{B}$ is asked to compile a constant. Then we need an instruction `PushV` (`IntVConst` **n**) to push an integer constant on the value stack. Its transition rule is quite simple:

(4.14)							
		`PushV (IntVConst` **n**`) :` **i**	**f**	**s**	**v**	**h**	**c**
	$\Longrightarrow$	**i**	**f**	**s**	**n** : **v**	**h**	**c**

Exercise 4.6. Implement the improved compilation scheme. Compare the performance of your implementation with that from before.

Exercise 4.7. Add a new rule to the $\mathcal{R}$ scheme to match a (full) application of `if`. You should be able to generate much better code than you get by calling the `if` primitive. Implement the change and measure the improvement in performance.

Exercise 4.8. Suppose we want to generalise our conditionals to deal with more general arithmetic comparisons, such as that required by

```
fib n = if (n < 2) 1 (fib (n-1) + fib (n-2))
```

What is required is a new instruction `Op Lt` which pops the top two items on the value stack, compares them, and pushes `1` or `0` onto the value stack depending on the result of the comparison. Now the `Cond` instruction can inspect this result.

Implement a family of such comparison instructions, and add special cases for them to the $\mathcal{B}$ scheme, in exactly the same way as for the other arithmetic operators. Test your improvement.

Exercise 4.9. In the previous exercise, you may have wondered why we did not modify the `Cond` instruction so that it had an extra 'comparison mode'. It could then compare the top two items on the value stack according to this mode, and act accordingly. Why did we not do this?

Hint: what would happen for programs like this?

```
multipleof3 x = ((x / 3) * 3) == x
f y = if (multipleof3 y) 0 1
```

The material of this section is discussed in [Argo 1989] and corresponds precisely to the improved G-machine compilation schemes discussed in Chapter 20 of [Peyton Jones 1987].

4.4 Mark 3: `let(rec)` expressions

At present the compiler cannot handle `let(rec)` expressions, a problem which we remedy in this section. Two main new ideas are introduced:

- We modify the `Take` instruction to allocate a frame with extra space to contain the `let(rec)`-bound variables, as well as the formal parameters.
- We introduce the idea of an *indirection closure*.

4.4.1 `let` expressions

When we compile a `let` expression, we must generate code to build new closures for the right-hand sides of the definitions. Where should these new closures be put? In order to treat `let(rec)`-bound names in the same way as argument names, they have to be put in the current frame[3]. This requires two modifications to the run-time machinery:

- The `Take` instruction should allocate a frame large enough to contain closures for all the `let` definitions which can occur during the execution of the supercombinator. The `Take` instruction must be modified to the form `Take` **t n**, where $\mathbf{t} \geq \mathbf{n}$. This instruction allocates a frame of size **t**, takes **n** closures from the top of the stack, and puts them into the first **n** locations of the frame.

[3] A hint of the material in this section is in [Wakeling and Dix 1989], but it is not fully worked out.

- We need a new instruction, `Move` **i a**, for moving a new closure **a** into slot **i** of the current frame. Here **a** is of type `timAMode` as for `Push` and `Enter`.

For example, the following definition:

```
f x = let y = f 3 in g x y
```

would compile to this code:

```
[ Take 2 1,
  Move 2 (Code [Push (IntConst 3), Enter (Label "f")]),
  Push (Arg 2),
  Push (Arg 1),
  Enter (Label "g")
]
```

Here is a slightly more elaborate example:

```
f x = let y = f 3
      in
      g (let z = 4 in h z) y
```

which generates the code:

```
[ Take 3 1,
  Move 2 (Code [Push (IntConst 3), Enter (Label "f")]),
  Push (Arg 2),
  Push (Code [Move 3 (IntConst 4), Push (Arg 3), Enter (Label "h")]),
  Enter (Label "g")
]
```

Notice the way that the initial `Take` allocates space for *all* the slots required by any of the closures in the body of the supercombinator.

Exercise 4.10. Write state transition rules for the new `Take` and `Move` instructions.

Next, we need to modify the compiler to generate the new `Take` and `Move` instructions. When we encounter a `let` expression we need to assign a free slot in the frame to each bound variable, so we need to keep track of which slots in the frame are in use and which are free. To do this, we add an extra parameter **d** to each compilation scheme, to record that the frame slots from $\mathbf{d}+1$ onwards are free, but that the slots from 1 to **d** might be occupied.

The remaining complication is that we need to discover the maximum value that **d** can take, so that we can allocate a big enough frame with the initial `Take` instruction. This requires each compilation scheme to return a pair: the compiled code, and the maximum value taken by **d**. The new compilation schemes are given in Figure 4.3. (In this figure, and subsequently, we use the notation $\mathbf{is}_1 \mathbin{+\!\!+} \mathbf{is}_2$ to denote the concatenation of the instruction sequences $\mathbf{is}_1$ and $\mathbf{is}_2$.)

$\mathcal{SC}[\![\mathbf{def}]\!]\ \rho$ is the TIM code for the supercombinator definition **def** compiled in environment ρ.

$$\mathcal{SC}[\![\mathbf{f}\ \mathbf{x}_1\ \ldots\ \mathbf{x_n}\ =\ \mathbf{e}]\!]\ \rho\ =\ \begin{array}{l}\texttt{Take}\ \mathbf{d}'\ \mathbf{n}\ :\ \mathbf{is} \\ \text{where } (\mathbf{d}', \mathbf{is}) = \mathcal{R}[\![\mathbf{e}]\!]\ \rho[\mathbf{x}_1 \mapsto \texttt{Arg}\ 1, \ldots, \mathbf{x_n} \mapsto \texttt{Arg}\ \mathbf{n}]\ \mathbf{n}\end{array}$$

$\mathcal{R}[\![\mathbf{e}]\!]\ \rho\ \mathbf{d}$ is a pair $(\mathbf{d}', \mathbf{is})$, where **is** is TIM code which applies the value of the expression **e** in environment ρ to the arguments on the stack. The code **is** assumes that the first **d** slots of the frame are occupied, and it uses slots $(\mathbf{d}+1 \ldots \mathbf{d}')$.

$$\mathcal{R}[\![\mathbf{e}]\!]\ \rho\ \mathbf{d}\ =\ \begin{array}{l}\mathcal{B}[\![\mathbf{e}]\!]\ \rho\ \mathbf{d}\ [\texttt{Return}] \\ \text{where } \mathbf{e} \text{ is an arithmetic expression or a number}\end{array}$$

$$\begin{array}{l}\mathcal{R}[\![\texttt{let}\ \mathbf{x}_1\texttt{=}\mathbf{e}_1\texttt{;}\ \ldots\texttt{;}\ \mathbf{x_n}\texttt{=}\mathbf{e_n}\ \texttt{in}\ \mathbf{e}]\!]\ \rho\ \mathbf{d} \\ \quad =\ (\mathbf{d}',\ [\texttt{Move}\ (\mathbf{d}+1)\ \mathbf{am}_1, \ldots,\ \texttt{Move}\ (\mathbf{d}+\mathbf{n})\ \mathbf{am_n}]\ +\!\!+\ \mathbf{is}) \\ \quad\quad \text{where } \begin{array}{lcl}(\mathbf{d}_1, \mathbf{am}_1) &=& \mathcal{A}[\![\mathbf{e}_1]\!]\ \rho\ (\mathbf{d}+\mathbf{n}) \\ (\mathbf{d}_2, \mathbf{am}_2) &=& \mathcal{A}[\![\mathbf{e}_2]\!]\ \rho\ \mathbf{d}_1 \\ \ldots && \\ (\mathbf{d_n}, \mathbf{am_n}) &=& \mathcal{A}[\![\mathbf{e_n}]\!]\ \rho\ \mathbf{d_{n-1}} \\ \rho' &=& \rho[\mathbf{x}_1 \mapsto \texttt{Arg}\ (\mathbf{d}+1), \ldots, \mathbf{x_n} \mapsto \texttt{Arg}\ (\mathbf{d}+\mathbf{n})] \\ (\mathbf{d}', \mathbf{is}) &=& \mathcal{R}[\![\mathbf{e}]\!]\ \rho'\ \mathbf{d_n}\end{array}\end{array}$$

$$\mathcal{R}[\![\mathbf{e}_1\ \mathbf{e}_2]\!]\ \rho\ \mathbf{d}\ =\ \begin{array}{l}(\mathbf{d}_2,\ \texttt{Push}\ \mathbf{am}\ :\ \mathbf{is}) \\ \text{where } \begin{array}{lcl}(\mathbf{d}_1, \mathbf{am}) &=& \mathcal{A}[\![\mathbf{e}_2]\!]\ \rho\ \mathbf{d} \\ (\mathbf{d}_2, \mathbf{is}) &=& \mathcal{R}[\![\mathbf{e}_1]\!]\ \rho\ \mathbf{d}_1\end{array}\end{array}$$

$$\mathcal{R}[\![\mathbf{a}]\!]\ \rho\ \mathbf{d}\ =\ \begin{array}{l}(\mathbf{d}',\ [\texttt{Enter}\ \mathbf{am}]) \\ \text{where } \mathbf{a} \text{ is a constant, supercombinator or local variable} \\ \text{and } (\mathbf{d}', \mathbf{am}) = \mathcal{A}[\![\mathbf{a}]\!]\ \rho\ \mathbf{d}\end{array}$$

$\mathcal{A}[\![\mathbf{e}]\!]\ \rho\ \mathbf{d}$ is a pair $(\mathbf{d}', \mathbf{am})$, where **am** is a TIM addressing mode for expression **e** in environment ρ. The code assumes that the first **d** slots of the frame are occupied, and it uses slots $(\mathbf{d}+1 \ldots \mathbf{d}')$.

$$\begin{array}{lcll}\mathcal{A}[\![\mathbf{x}]\!]\ \rho\ \mathbf{d} &=& (\mathbf{d},\ \rho\ \mathbf{x}) & \text{where } \mathbf{x} \text{ is bound by } \rho \\ \mathcal{A}[\![\mathbf{n}]\!]\ \rho\ \mathbf{d} &=& (\mathbf{d},\ \texttt{IntConst}\ \mathbf{n}) & \text{where } \mathbf{n} \text{ is an integer constant} \\ \mathcal{A}[\![\mathbf{e}]\!]\ \rho\ \mathbf{d} &=& (\mathbf{d}',\ \texttt{Code}\ \mathbf{is}) & \text{otherwise} \\ && \text{where } (\mathbf{d}', \mathbf{is}) = \mathcal{R}[\![\mathbf{e}]\!]\ \rho\ \mathbf{d} &\end{array}$$

Figure 4.3: Compilation schemes for `let` expressions

In the $\mathcal{SC}$ scheme you can see how the maximum frame size $\mathbf{d'}$, returned from the compilation of the supercombinator body, is used to decide how large a `Take` to perform. In the `let` expression case of the $\mathcal{R}$ scheme, for each definition we generate an instruction **`Move i a`**, where $\mathbf{i}$ is the number of a free slot in the current frame, and $\mathbf{a}$ is the result of compiling $\mathbf{e}$ with the $\mathcal{A}$ scheme. Notice the way in which the compilation of each right-hand side is given the index of the last slot occupied by the previous right-hand side, thus ensuring that all the right-hand sides use different slots.

Exercise 4.11. Implement the changes described in this section: add the new instructions to the `instruction` type, add new cases to `step` and `showInstruction` to handle them, and implement the new compilation schemes.

Exercise 4.12. Consider the program

```
f x y z = let p = x+y in p+x+y+z
main = f 1 2 3
```

In the absence of `let` expressions, it would have to be written using an auxiliary function, like this:

```
f' p x y z = p+x+y+z
f x y z = f' (x+y) x y z
main = f 1 2 3
```

Compare the code generated by these two programs, and measure the difference in store consumed and steps executed. What is the main saving obtained by implementing `let` expressions directly?

4.4.2 `letrec` expressions

What needs to be done to handle `letrec` expressions as well? At first it seems very easy: the `letrec` case for the $\mathcal{R}$ scheme is exactly the same as the `let` case, except that we need to replace ρ by ρ' in the definitions of the $\mathbf{am_i}$. This is because the $\mathbf{x_i}$ are in scope in their own right-hand sides.

Exercise 4.13. Implement this extra case in `compileR`, and try it out on the program

```
f x = letrec p = if (x==0) 1 q ;
             q = if (x==0) p 2
      in p+q
main = f 1
```

Make sure you understand the code which is generated, and test it.

Unfortunately there is a subtle bug in this implementation! Consider the code generated from:

```
f x = letrec a = b ;
             b = x
      in a
```

which is as follows:

```
[Take 3 1, Move 2 (Arg 3), Move 3 (Arg 1), Enter (Arg 2)]
```

The closure for `b` is copied by the first `Move` before it is assigned by the second `Move`!

There are two ways out of this. The first is to declare that this is a silly program; just replace `b` by `x` in the scope of the binding for `b`. But there is a more interesting approach which will be instructive later, and which allows even silly programs like the one above to work. Suppose we generate instead the following code for the first `Move`:

```
Move 2 (Code [Enter (Arg 3)])
```

Now everything will be fine: slot 3 will be assigned before the `Enter (Arg 3)` gets executed. In fact you can think of the closure `([Enter (Arg 3)], f)` as an *indirection* to slot 3 of frame `f`.

This code can be obtained by modifying the `let(rec)` case of the $\mathcal{R}$ compilation scheme, so that it records an *indirection addressing mode* in the environment for each variable bound by the `let(rec)`. Referring to Figure 4.3, the change required is to the rule for `let` in the $\mathcal{R}$ scheme, where the definition of ρ' becomes

$$\begin{aligned} \rho' = \ & \rho[\mathbf{x}_1 \mapsto \mathcal{I}[\![\mathbf{d}+1]\!], \ldots, \mathbf{x_n} \mapsto \mathcal{I}[\![\mathbf{d}+\mathbf{n}]\!]] \\ & \text{where } \mathcal{I}[\![\mathbf{d}]\!] = \texttt{Code [Enter (Arg } \mathbf{d}\texttt{)]} \end{aligned}$$

Of course, this is rather conservative: it returns an indirection in lots of cases where it is not necessary to do so, but the resulting code will still work fine, albeit less efficiently.

Exercise 4.14. Modify `compileR` to implement this idea, and check that it generates correct code for the above example. In modifying `compileR` use the auxiliary function `mkIndMode` (corresponding to the $\mathcal{I}$ scheme) to generate the indirection addressing modes in the new environment:

```
> mkIndMode :: num -> timAMode
> mkIndMode n = Code [Enter (Arg n)]
```

4.4.3 Reusing frame slots†

At present, every definition on the right-hand side of a supercombinator definition gets its own private slot in the frame. Sometimes you may be able to figure out that you can safely share slots between different `let(rec)`s. For example, consider the definition

```
f x = if x (let ... in ...) (let ... in ...)
```

Now it is plain that only one of the two `let` expressions can ever be evaluated, so it would be perfectly safe to use the same slots for their definitions.

Similarly, in the expression $\mathbf{e}_1$ `+` $\mathbf{e}_2$, any `let(rec)` slots used during the evaluation of $\mathbf{e}_1$ will be finished with by the time $\mathbf{e}_2$ is evaluated (or vice versa if `+` happened to evaluate its arguments in the reverse order), so any `let(rec)`-bound variables in $\mathbf{e}_1$ can share slots with those in $\mathbf{e}_2$.

Exercise 4.15. Modify your compiler to spot this and take advantage of it.

4.4.4 Garbage collection†

In Section 4.2.6 we remarked that it would be desirable to record which frame slots were used by a code sequence, so that space leaks can be avoided. If `Take` does not initialise the extra frame slots which it allocates, there is a danger that the garbage collector will treat the contents of these uninitialised slots as valid pointers, with unpredictable results. The easiest solution is to initialise all slots, but this is quite expensive. It is better to adopt the solution of Section 4.2.6, and record with each code sequence the list of slots which should be retained. Uninitialised slots will then never be looked at by the garbage collector.

4.5 Mark 4: Updating

So far we have been performing tree reduction not graph reduction, because we repeatedly evaluate shared redexes. It is time to fix this. Figuring out exactly how the various ways of performing TIM updates work is a little tricky, but at least we have a prototype implementation so that our development will be quite concrete.

4.5.1 The basic technology

The standard template instantiation machine and the G-machine perform an update after every reduction. (The G-machine has a few optimisations for tail calls, but the principle is the same.) Because TIM is a *spineless* machine, its updating technique has to be rather different. The key idea is this:

> *Updates are not performed after each reduction, as the G-machine does. Instead, when evaluation of a closure is started (that is, when the closure is entered), the following steps are taken:*
>
> - *The current stack, and the address of the closure being entered, are pushed onto the* dump*, a new component of the machine state.*
> - *A 'mouse-trap' is set up which is triggered when evaluation of the closure is complete.*
> - *Evaluation of the closure now proceeds normally, starting with an empty stack.*
> - *When the mouse-trap is triggered, the closure is updated with its normal form, and the old stack is restored from the dump.*
>
> *The 'mouse-trap' is the following. Since the evaluation of the closure is carried out on a new stack, the evaluation must eventually grind to a halt, because a* `Return` *instruction finds an empty stack, or a supercombinator is being applied to too few arguments. At this point the expression has reached (head) normal form, so an update should be performed.*

To begin with, let us focus on updating a closure whose value is of integer type. We will arrange that just before the closure is entered, a new instruction `PushMarker x` is executed, which sets up the update mechanism by pushing some information on the dump. Specifically, `PushMarker x` pushes onto the dump[4]:

- The current stack.
- The current frame pointer, which points to the frame containing the closure to be updated.
- The index, $\mathbf{x}$, of the closure to be updated within the frame.

Now that it has saved the current stack on the dump, `PushMarker` continues with an empty stack. Here is its state transition rule:

$$
(4.15)\qquad
\begin{array}{|lrccccccc|}
\hline
 & \texttt{PushMarker}\ \mathbf{x} : \mathbf{i} & \mathbf{f} & \mathbf{s} & \mathbf{v} & & \mathbf{d} & \mathbf{h} & \mathbf{c} \\
\Longrightarrow & \mathbf{i} & \mathbf{f} & [] & \mathbf{v} & (\mathbf{f},\mathbf{x},\mathbf{s}) : & \mathbf{d} & \mathbf{h} & \mathbf{c} \\
\hline
\end{array}
$$

Some while later, evaluation of the closure will be complete. Since its value is an integer, its value will be on top of the value stack, and a `Return` instruction will be executed in the expectation of returning to the continuation on top of the stack. But the stack will be empty at this point! This is what triggers the update: the dump is popped, the update is performed, and the `Return` instruction is re-executed with the restored stack. This action is described by the following transition rules:

$$
(4.16)\qquad
\begin{array}{|llrcrccc|}
\hline
 & [\texttt{Return}] & \mathbf{f} & [] & \mathbf{n} : \mathbf{v} & (\mathbf{f_u},\mathbf{x},\mathbf{s}) : \mathbf{d} & \mathbf{h} & \mathbf{c} \\
\Longrightarrow & [\texttt{Return}] & \mathbf{f} & \mathbf{s} & \mathbf{n} : \mathbf{v} & \mathbf{d} & \mathbf{h'} & \mathbf{c} \\
 & \multicolumn{7}{l|}{\mathbf{h'} = \mathbf{h}[\mathbf{f_u} : \langle \ldots, \mathbf{d_{x-1}}, (\texttt{intCode}, \mathbf{n}), \mathbf{d_{x+1}}, \ldots \rangle]} \\
\hline
 & [\texttt{Return}] & \mathbf{f} & (\mathbf{i},\mathbf{f'}) : \mathbf{s} & \mathbf{n} : \mathbf{v} & \mathbf{d} & \mathbf{h} & \mathbf{c} \\
\Longrightarrow & \mathbf{i} & \mathbf{f'} & \mathbf{s} & \mathbf{n} : \mathbf{v} & \mathbf{d} & \mathbf{h} & \mathbf{c} \\
\hline
\end{array}
$$

The first rule describes the update of the $\mathbf{x}$th closure in frame $\mathbf{f_u}$, with the closure $(\texttt{intCode}, \mathbf{n})$. This is a closure which when entered immediately pushes $\mathbf{n}$ onto the value stack and returns (see Section 4.3.1). Notice that the `Return` instruction is retried, in case there is a further update to perform; this is indicated by the fact that the code sequence in the right-hand side of the rule is still [`Return`].

The second rule is just Rule 4.11 written out again. It covers the case when there is no update to be performed; the continuation on top of the stack is loaded into the program counter and current frame pointer.

4.5.2 Compiling `PushMarker` instructions

The execution of the `PushMarker` and `Return` instructions is thus quite straightforward, but the tricky question is: where should the compiler plant `PushMarker` instructions? To answer this we have to recall the motivation for the whole updating exercise: it is to ensure that each redex is only evaluated once, by overwriting the redex with its value once it has been evaluated. In TIM, a 'redex' is a closure. The key insight is this:

[4]In papers about TIM this operation is often called 'pushing an update marker', because the stack is 'marked' so that the attempt to use arguments below the 'marker' will trigger an update.

we have to be very careful when copying closures, because once two copies exist there is no way we can ever share their evaluation.

For example, consider the following function definitions:

```
g x = h x x
h p q = q - p
```

At present we will generate the following code for `g`:

```
[ Take 1 1, Push (Arg 1), Push (Arg 1), Enter (Label "h") ]
```

The two `Push Arg` instructions will each take a copy of the same closure, and `h` will subsequently evaluate each of them independently.

What we really want to do is to push not a *copy* of the closure for `x`, but rather a *pointer to* it. Recalling the idea of an *indirection* closure from Section 4.4.2, this is easily done, by replacing `Push (Arg 1)` with `Push (Code [Enter (Arg 1)])`.

This gets us half-way; we are not duplicating the closure, but we still are not updating it. But now it is easy! All we need to is to precede the `Enter (Arg 1)` instruction with `PushMarker 1`, thus:

```
Push (Code [PushMarker 1, Enter (Arg 1)])
```

That is, just before entering the shared closure, we set up the update mechanism which will cause it to be updated when its evaluation is complete.

The addressing mode (`Code` [`PushMarker` **n**, `Enter` (`Arg` **n**)]) is called an *updating indirection* to the **n**th closure of the frame, because it is an indirection which will cause an update to take place. Exactly the same considerations apply to entering an argument (rather than pushing it on the stack). An updating indirection to the argument must be entered, rather than the argument itself: `Enter (Arg 1)` must be replaced by `Enter (Code [PushMarker 1, Enter (Arg 1)])`.

The changes to the compilation schemes are simple. Only the $\mathcal{SC}$ and $\mathcal{R}$ schemes are affected, and they are both affected in the same way: where they build an environment, they should bind each variable to an updating indirection addressing mode. For example, in the `let(rec)` case of the $\mathcal{R}$ scheme, we now use the following definition for ρ' (cf. Figure 4.3):

$$\begin{aligned} \rho' = \ & \rho[\mathbf{x_1} \mapsto \mathcal{J}[\![\mathbf{d}+1]\!], \ldots, \mathbf{x_n} \mapsto \mathcal{J}[\![\mathbf{d}+\mathbf{n}]\!]] \\ & \text{where } \mathcal{J}[\![\mathbf{d}]\!] = \texttt{Code}\ [\texttt{PushMarker}\ \mathbf{d},\ \texttt{Enter}\ (\texttt{Arg}\ \mathbf{d})] \end{aligned}$$

4.5.3 Implementing the updating mechanism

To implement the new updating mechanism, we need to make the following changes:

- Give a type definition for the dump. It is just a stack of triples, represented as a list, and initialised to be empty:

```
> timDump == [(framePtr,  || The frame to be updated
```

```
>              num,       || Index of slot to be updated
>              timStack)  || Old stack
>            ]
> initialDump = []
```

- Add the PushMarker instruction to the instruction type, with appropriate modifications to showInstruction.
- Add a new case to step for the PushMarker instruction, and modify the case for Return.
- Modify compileSC and the ELet case of compileR to build environments which bind each variable to an updating indirection addressing mode. Use the function mkUpdIndMode to implement the $\mathcal{J}$ scheme:

```
> mkUpdIndMode :: num -> timAMode
> mkUpdIndMode n = Code [PushMarker n, Enter (Arg n)]
```

Exercise 4.16. Implement the updating mechanism as described.

When running the new system on some test programs, you should be able to watch PushMarker adding update information to the dump, and Return performing the updates. Here is one possible test program:

```
f x = x + x
main = f (1+2)
```

The evaluation of (1+2) should only happen once!

Exercise 4.17. Here is an easy optimisation you can perform. For a function such as:

```
compose f g x = f (g x)
```

you will see that the code for compose is like this:

```
compose:        Take 3 3
                Push (Code [...])
                Enter (Code [PushMarker 1, Enter (Arg 1)])
```

where the [...] is code for (g x). The final instruction enters an updating indirection for f. But it is a fact that

Enter (Code **i**)   is equivalent to   **i**

(This follows immediately from Rule 4.8.) So the equivalent code for compose is

```
compose:        Take 3 3
                Push (Code [...])
                PushMarker 1
                Enter (Arg 1)
```

Implement this optimisation. Much the nicest way to do this is by replacing all expressions in the compiler of the form [Enter **e**] with (mkEnter **e**), where mkEnter is defined like this:

```
> mkEnter :: timAMode -> [instruction]
> mkEnter (Code i) = i
> mkEnter other_am = [Enter other_am]
```

`mkEnter` is an 'active' form of the `Enter` constructor, which checks for a special case before generating an `Enter` instruction.

There are a number of other improvements we can make to this scheme, and we will study them in the following sections.

4.5.4 Problems with updating indirections

While it is simple enough, this updating mechanism is horribly inefficient. There are two main problems, which were first distinguished by Argo [Argo 1989]. The first problem is that of *identical updates.* Consider the program given in the previous section:

```
f x = x+x
main = f (1+2)
```

For each use of `x`, `f` will enter an updating closure to its argument `x`. The first time, `x` will be updated with its value. The second time, a second (and entirely redundant) update will take place, which overwrites the `x` with its value again. You should be able to watch this happening as you execute the example on your implementation.

In this case, of course, a clever compiler could spot that `x` was sure to be evaluated, and just copy `x` instead of entering an indirection to it. But this complicates the compiler, and in general may be impossible to spot. For example, suppose `f` was defined like this:

```
f x = g x x
```

Unless `f` analyses `g` to discover in which order the different `x`'s are evaluated (and in general there may be no one answer to this question), it has to be pessimistic and push updating indirections as arguments to `g`.

The second problem is that of *indirection chains.* Consider the program

```
g x = x+x
f y = g y
main = f (1+2)
```

Here, `f` passes to `g` an updating indirection to its argument `y`; but `g` enters an updating indirection to its argument `x`. Thus `g` enters an indirection to an indirection. In short, chains of indirections build up, because *an indirection is added every time an argument is passed on as an argument to another function.* Just imagine how many indirections to `m` could build up in the following tail-recursive function!

```
horrid n m = if (n=0) m (horrid (n-1) m)
```

We will not solve these problems yet. Instead, the next section shows how to deal with updates for `let(rec)`-bound variables, and why these two problems do not arise. This points the way to a better solution for supercombinator arguments as well.

4.5.5 Updating shared `let(rec)`-bound variables

So far we have assumed that `let(rec)`-bound variables are updated in exactly the same way as supercombinator arguments, by always using an updating indirection addressing mode for them. For example, consider the following supercombinator definition:

```
        f x = let y = ...
              in
              g y y
```

where the '`...`' stands for an arbitrary right-hand side for `y`. Treating `y` just the same as `x`, we will generate this code for `f`:

```
f:      Take 2 1                                    -- Frame with room for y
        Move 2 (Code [...code for y...])            -- Closure for y
        Push (Code [PushMarker 2, Enter (Arg 2)])   -- Indirection to y
        Push (Code [PushMarker 2, Enter (Arg 2)])   -- Indirection to y
        Enter (Label "g")
```

where the '`...code for y...`' stands for the code generated from `y`'s right-hand side. This code suffers from the identical-update problem outlined earlier.

But a much better solution is readily available. Suppose we generate the following code for `f` instead:

```
f:      Take 2 1                            -- Frame with room for y
        Move 2 (Code (PushMarker 2 :
                     [...code for y...]))  -- Closure for y
        Push (Code [Enter (Arg 2)])         -- Non-updating indirection to y
        Push (Code [Enter (Arg 2)])         -- Indirection to y
        Enter (Label "g")
```

The `PushMarker` instruction has moved from the *uses* of `y` to its *definition*. The closure for `y` built by the `Move` instruction is now a *self-updating closure*; that is, when entered it will set up the update mechanism which will update itself. Once this has happened, it will never happen again because the pointer to the code with the `PushMarker` instruction has now been overwritten!

In general, the idea is this:

- Use a self-updating closure for the right-hand side of a `let(rec)` binding, by beginning the code with a `PushMarker` instruction.
- Use ordinary (non-updating) indirection addressing modes when pushing `let(rec)`-bound variables onto the stack. We still need to use indirections, rather than taking a copy of the closure because, until it is updated, copying it would give rise to duplicated work.

The modifications needed to implement this idea are:

$\mathcal{U}[\![\mathbf{e}]\!]\ \mathbf{u}\ \rho\ \mathbf{d}$ is a pair $(\mathbf{d}', \mathbf{am})$, where **am** is a TIM addressing mode for expression **e** in environment ρ. If the closure addressed by **am** is entered, it will update slot **u** of the current frame with its normal form. The code assumes that the first **d** slots of the frame are occupied, and it uses slots $(\mathbf{d}+1 \ldots \mathbf{d}')$.

$$\begin{array}{lcl} \mathcal{U}[\![\mathbf{e}]\!]\ \mathbf{u}\ \rho\ \mathbf{d} & = & (\mathbf{d}',\ \texttt{Code}\ (\texttt{PushMarker}\ \mathbf{u} : \mathbf{is})) \\ & & \text{where } (\mathbf{d}', \mathbf{is}) = \mathcal{R}[\![\mathbf{e}]\!]\ \rho\ \mathbf{d} \end{array}$$

$$\begin{array}{l} \mathcal{R}[\![\texttt{let}\ \mathbf{x_1}\texttt{=}\mathbf{e_1}\texttt{;}\ \ldots\texttt{;}\ \mathbf{x_n}\texttt{=}\mathbf{e_n}\ \texttt{in}\ \mathbf{e}]\!]\ \rho\ \mathbf{d} \\ \quad = \ (\mathbf{d}',\ [\texttt{Move}\ (\mathbf{d}+1)\ \mathbf{am}_1,\ \ldots, \texttt{Move}\ (\mathbf{d}+\mathbf{n})\ \mathbf{am_n}]\ +\!\!+\ \mathbf{is}) \\ \qquad \text{where} \begin{array}[t]{lcl} (\mathbf{d}_1, \mathbf{am}_1) & = & \mathcal{U}[\![\mathbf{e}_1]\!]\ (\mathbf{d}+1)\ \rho\ (\mathbf{d}+\mathbf{n}) \\ (\mathbf{d}_2, \mathbf{am}_2) & = & \mathcal{U}[\![\mathbf{e}_2]\!]\ (\mathbf{d}+2)\ \rho\ \mathbf{d}_1 \\ \ldots & & \\ (\mathbf{d_n}, \mathbf{am_n}) & = & \mathcal{U}[\![\mathbf{e_n}]\!]\ (\mathbf{d}+\mathbf{n})\ \rho\ \mathbf{d_{n-1}} \\ \rho' & = & \rho[\mathbf{x}_1 \mapsto \mathcal{I}[\![\mathbf{d}+1]\!], \ldots, \mathbf{x_n} \mapsto \mathcal{I}[\![\mathbf{d}+\mathbf{n}]\!]\,] \\ & & \text{where } \mathcal{I}[\![\mathbf{d}]\!] = \texttt{Code}\ [\texttt{Enter}\ (\texttt{Arg}\ \mathbf{d})] \\ (\mathbf{d}', \mathbf{is}) & = & \mathcal{R}[\![\mathbf{e}]\!]\ \rho'\ \mathbf{d_n} \end{array} \end{array}$$

The `letrec` case is similar, except that ρ' is passed to the calls to $\mathcal{U}[\![]\!]$ instead of ρ.

Figure 4.4: The $\mathcal{U}$ compilation scheme, and revised $\mathcal{R}$ rule for `let`

- Modify the $\mathcal{R}$ scheme so that it generates *non-updating* indirections for `let(rec)`-bound variables; that is, it builds the new environment using the $\mathcal{I}$ scheme rather than $\mathcal{J}$. (For the present, $\mathcal{SC}$ should continue to generate *updating* indirections, using $\mathcal{J}$, for supercombinator arguments.)

- Modify the $\mathcal{R}$ scheme for `let(rec)` expressions, so that it generates a `PushMarker` instruction at the start of the code for every right-hand side. This is most conveniently done by creating a new compilation scheme, the $\mathcal{U}$ scheme (see Figure 4.4), which is used in the `let(rec)` case of the $\mathcal{R}$ scheme to compile the right-hand sides of definitions. $\mathcal{U}$ needs a extra argument to tell it which slot in the current frame should be updated, and uses this argument to generate an appropriate `PushMarker` instruction. Figure 4.4 also gives the revised `let` equation for the $\mathcal{R}$ scheme. The modification to the `letrec` case is exactly analogous.

Exercise 4.18. Try out this idea and measure its effectiveness in terms of how many steps are saved.

Exercise 4.19. Consider the expression

```
let x = 3 in x+x
```

In this case, the right-hand side of the `let` expression is already in normal form, so there is no point in the $\mathcal{U}$ scheme generating a `PushMarker` instruction. Rather, $\mathcal{U}$ can simply return an `IntConst` addressing mode for this case.

Modify the $\mathcal{U}$ compilation scheme, and the corresponding `compileU` function, and confirm that the modification works correctly.

4.5.6 Eliminating indirection chains

The idea of the previous section shows how to eliminate identical updates for `let(rec)`-bound variables. In this section we show how to extend the idea to eliminate identical updates for supercombinator arguments as well and, at the same time, to eradicate the indirection chain problem. The idea was first proposed by Argo [Argo 1989].

We start with indirection chains. As noted earlier, indirection chains build up because a supercombinator has to assume that it must not copy any of its argument closures, so if it uses them more than once it had better use indirections. This gives rise to indirection chains because often the argument closure is an indirection already, and it would be perfectly safe to copy it.

This suggests an alternative strategy:

> *adopt the convention that every argument closure must be freely copyable without loss of sharing.*

This calling convention is clearly convenient for the called function, but how can the caller ensure that it is met? An argument is either:

- a constant, whose closure is freely copyable;
- a supercombinator, which has the same property;
- a `let(rec)`-bound variable, which also can be freely copied (using the ideas of the previous section);
- an argument to the current supercombinator, which is freely copyable because of our new convention;
- a non-atomic expression, whose closure (as things stand) is *not* freely copyable.

It follows that all we have to do to adopt this new calling convention is find some way of passing non-atomic arguments as freely copyable closures. For example, consider the expression

```
f (factorial 20)
```

How can the argument `(factorial 20)` be passed as a freely copyable closure. The solution is simple: transform the expression to the following equivalent form:

```
let arg = factorial 20 in f arg
```

The `let` expression will allocate a slot in the current frame for the closure for `(factorial 20)`, will put a self-updating closure in it, and an (ordinary) indirection to this closure will be passed to `f`. (Notice that this transformation need only be carried out if the closure passed to `f` might

$$\mathcal{R}[\![\mathbf{e}\ \mathbf{a}]\!]\ \rho\ \mathbf{d} = (\mathbf{d}_1,\ \texttt{Push}\ (\mathcal{A}[\![\mathbf{a}]\!]\ \rho)\ :\ \mathbf{is})$$
$$\text{where } \mathbf{a} \text{ is a supercombinator, local variable, or constant}$$
$$(\mathbf{d}_1, \mathbf{is}) = \mathcal{R}[\![\mathbf{e}]\!]\ \rho\ \mathbf{d}$$

$$\mathcal{R}[\![\mathbf{e_{fun}}\ \mathbf{e_{arg}}]\!]\ \rho\ \mathbf{d} = (\mathbf{d}_2,\ \texttt{Move}\ (\mathbf{d}+1)\ \mathbf{am_{arg}} : \texttt{Push}\ (\texttt{Code}\ [\texttt{Enter}\ (\texttt{Arg}\ (\mathbf{d}+1))]) : \mathbf{is_{fun}})$$
$$\text{where } (\mathbf{d}_1, \mathbf{am_{arg}}) = \mathcal{U}[\![\mathbf{e_{arg}}]\!]\ (\mathbf{d}+1)\ \rho\ (\mathbf{d}+1)$$
$$(\mathbf{d}_2, \mathbf{is_{fun}}) = \mathcal{R}[\![\mathbf{e_{fun}}]\!]\ \rho\ \mathbf{d}_1$$

$$\mathcal{A}[\![\mathbf{n}]\!]\ \rho = \texttt{IntConst}\ \mathbf{n} \quad \text{where } \mathbf{n} \text{ is a number}$$
$$\mathcal{A}[\![\mathbf{x}]\!]\ \rho = \rho\ \mathbf{x} \quad \text{where } \mathbf{x} \text{ is bound by } \rho$$

Figure 4.5: Modifications to $\mathcal{R}$ and $\mathcal{A}$ for copyable arguments

be entered more than once. There is an opportunity here for a global sharing analysis to be used to generate more efficient code.)

Once this transformation is done we can freely copy argument closures, though we must still use (non-updating) indirections for `let(rec)`-bound closures. No indirection chains will build up, nor will identical updates take place.

It is interesting to reflect on what has happened. At first it appeared as though TIM would allocate much less heap than a G-machine, because the entire allocation for a supercombinator call was the frame required to hold its arguments. However, using our new updating techniques, we see that every sub-expression within the supercombinator body requires a slot in the frame to hold it. Similarly, since most supercombinator arguments are now indirections, TIM is behaving quite like the G-machine which passes pointers to arguments rather than the arguments themselves. So the problems of lazy updating have forced TIM to become more G-machine-like.

We have presented the technique as a program transformation which introduces a `let` expression for every argument expression, but doing so is somewhat tiresome because it involves inventing new arbitrary variable names. It is easier to write the new compilation schemes more directly. The main alteration is to the application case of the $\mathcal{R}$ scheme, which is given in Figure 4.5. The first equation deals with the case where the argument to the application is an atomic expression (variable or constant), using the $\mathcal{A}$ scheme to generate the appropriate addressing mode as before. The second equation deals with the case where the argument is a compound expression; it initialises the next free slot in the frame with a self-updating closure for the argument expression, and pushes an indirection to this closure.

The $\mathcal{A}$ scheme, also given in Figure 4.5, now has one case fewer than before, because it is only invoked with an atomic expression (variable or constant) as its argument. For the same reason, it no longer needs to take $\mathbf{d}$ as an argument and return it as a result, because it never uses any frame slots.

Exercise 4.20. Implement this revised scheme, and measure the difference in performance from the previous version.

4.5.7 Updating partial applications

So far we have successfully dealt with the update of closures whose value is an integer. When a `Return` instruction finds an empty stack, it performs an update and pops a new stack from the dump.

But there is another instruction which consumes items from the stack, namely `Take`. What should happen if a `Take` instruction finds fewer items on the stack than it requires? For example, consider the program

```
add a b = a+b
twice f x = f (f x)
g x = add (x*x)
main = twice (g 3) 4
```

When `twice` enters `f` it will do so via an indirection, which will set up an update for `f`. In this example, `f` will be bound to `(g 3)`, which evaluates to a partial application of `add` to one argument. The `Take 2` instruction at the beginning of the code for `add` will discover that there is only one argument on the stack, which indicates that an update should take place, overwriting the closure for `(g x)` with one for `(add (x*x))`.

In general:

> *when a **`Take`** instruction finds too few arguments on the stack, it should perform an update on the closure identified by the top item on the dump, glue the items on the current stack on top of the stack recovered from the dump, and retry the **`Take`** instruction (in case another update is required).*

The `Take` instruction is already complicated enough, and now it has a further task to perform. To avoid `Take` getting too unwieldy, we split it into two instructions: `UpdateMarkers`, which performs the check as to whether there are enough arguments, and `Take` which actually builds the new frame. An `UpdateMarkers n` instruction always immediately precedes every `Take t n` instruction.

The transition rule for `Take` is therefore unchanged. The rules for `UpdateMarkers` are as follows:

$$
(4.17)\quad
\begin{array}{|llllllllll|}
\hline
 & \texttt{UpdateMarkers}\ \mathbf{n} : \mathbf{i} & \mathbf{f} & \mathbf{c_1 : \ldots : c_m : s} & \mathbf{v} & & \mathbf{d} & \mathbf{h} & \mathbf{c} & \\
\Longrightarrow & \mathbf{i} & \mathbf{f} & \mathbf{c_1 : \ldots : c_m : s} & \mathbf{v} & & \mathbf{d} & \mathbf{h} & \mathbf{c} & \\
 & \text{where } \mathbf{m \geq n} & & & & & & & & \\
\hline
 & \texttt{UpdateMarkers}\ \mathbf{n} : \mathbf{i} & \mathbf{f} & \mathbf{c_1 : \ldots : c_m : []} & \mathbf{v} & \mathbf{(f_u, x, s)} : & \mathbf{d} & \mathbf{h} & \mathbf{c} & \\
\Longrightarrow & \texttt{UpdateMarkers}\ \mathbf{n} : \mathbf{i} & \mathbf{f} & \mathbf{c_1 : \ldots : c_m : s} & \mathbf{v} & & \mathbf{d} & \mathbf{h'} & \mathbf{c} & \\
 & \text{where}\quad \mathbf{m < n} & & & & & & & & \\
 & \quad\quad \mathbf{h' = h[f_u : \langle \ldots, d_{x-1}, (i', f'), d_{x+1}, \ldots \rangle]} & & & & & & & & \\
\hline
\end{array}
$$

The first rule deals with the case where there are enough arguments, so that `UpdateMarkers` does nothing. The second deals with the other case, where an update needs to take place; the appropriate closure is updated, the current stack is glued on top of the old one, and the `UpdateMarkers` is retried.

In this rule, $\mathbf{i}'$ and $\mathbf{f}'$ are the code pointer and frame pointer which overwrite the target closure, but so far we have not specified just what values they should take. The way to figure out what they should be is to ask the question: what should happen when the closure $(\mathbf{i}', \mathbf{f}')$ is entered? This closure represents the partial application of the supercombinator to the arguments $\mathbf{c}_1, \ldots, \mathbf{c_m}$. Hence, when it is entered, it should push $\mathbf{c}_1, \ldots, \mathbf{c_m}$, and then jump to the code for the supercombinator. It follows that

- $\mathbf{f}'$ must point to a newly allocated frame $\langle \mathbf{c}_1, \ldots, \mathbf{c_m} \rangle$.
- $\mathbf{i}'$ must be the code sequence

$$\texttt{Push (Arg } \mathbf{m}\texttt{)} : \ldots : \texttt{Push (Arg } 1\texttt{)} : \texttt{UpdateMarkers } \mathbf{n} : \mathbf{i}$$

  Here, the `Push` instructions place the arguments of the partial application onto the stack, the `UpdateMarkers` instruction checks for any further updates that need to take place, and $\mathbf{i}$ is the rest of the code for the supercombinator.

Exercise 4.21. Implement the `UpdateMarkers` instruction, and modify the compiler to place one before each `Take` instruction. Test your implementation before and after the modification on the following program. The program uses higher-order functions to implement pairs (Section 2.8.3). The pair `w` is shared, and evaluates to a partial application of the `pair` function.

```
pair x y f = f x y
fst p = p K
snd p = p K1
main = let w = pair 2 3
       in (fst w) * (snd w)
```

You should see `w` being updated with the partial application for `(pair 2 3)`. To make it a little more convincing, you could make the right-hand side of `w` involve a little more computation: for example

```
main = let w = if (2*3 > 4) (pair 2 3) (pair 3 2)
       in (fst w) * (snd w)
```

Exercise 4.22. Just as `Take 0 0` does nothing, `UpdateMarkers 0` does nothing. Modify `compileSC` so that it omits both of these instructions when appropriate. (This is a simple extension of Exercise 4.3.)

There are a few other points worth noticing:

- In a real implementation, the code $\mathbf{i}'$ would not be manufactured afresh whenever an update takes place, as the rule appears to say. Instead, the code for the supercombinator $\mathbf{i}$ can be preceded by a sequence of `Push` instructions, and the code pointer for a partial application can just point into the appropriate place in the sequence.
- The `UpdateMarkers` rule duplicates the closures $\mathbf{c}_1, \ldots, \mathbf{c_m}$. This is fine now that supercombinator arguments are freely copyable, a modification we introduced in Section 4.5.6. Prior to that modification, making such a copy would have risked duplicating a redex, so instead the `UpdateMarkers` rule would have been further complicated with indirections. It is for this reason that the introduction of `UpdateMarkers` has been left so late.

- Suppose we are compiling code for the expression (`f` $\mathbf{e}_1$ $\mathbf{e}_2$), where `f` is known to be a supercombinator of 2 (or fewer) arguments. In this case, the `UpdateMarkers` instruction at the start of `f` will certainly do nothing, because the stack is sure to be deep enough to satisfy it. So when compiling a call to a supercombinator applied to all its arguments (or more) we can enter its code *after* the `UpdateMarkers` instruction.

  Many of the function applications in a typical program are saturated applications of known supercombinators, so this optimisation is frequently applicable.

4.6 Mark 5: Structured data

In this section we will study how to add algebraic data types to TIM. It is possible to implement data structures without any of the material of this section, using higher-order functions as described in Section 2.8.3; but it is rather inefficient to do so. Instead, we will develop the approach we used for arithmetic to be able to handle more general data structures.

4.6.1 The general approach

Consider the function `is_empty`, which returns 1 if its argument is an empty list, and 0 if not. It is given in the context of a program which applies it to a singleton list.

```
is_empty xs = case xs of
                <1>      -> 1
                <2> y ys -> 0

cons a b = Pack{2,2} a b
nil = Pack{1,0}

main = is_empty (cons 1 nil)
```

Recall from Section 1.1.4 that constructors are denoted by `Pack{`**tag**`,`**arity**`}`. In this program, which manipulates lists, the empty list constructor `nil` has tag 1 and arity 0, while the list constructor `cons` has tag 2 and arity 2. Pattern matching is performed only by `case` expressions; nested patterns are matched by nested `case` expressions.

We consider first what code we should generate for a `case` expression. Just as arithmetic operators require their arguments to be evaluated, a `case` expression requires an expression, `xs` in the `is_empty` example, to be evaluated. After this, a multi-way jump can be taken depending on the tag of the object returned. Taking a similar approach to the one we used for arithmetic operators suggests the following conventions:

- To evaluate a closure representing a data object, a continuation is pushed onto the argument stack, and the closure is entered.
- When it is evaluated to (head) normal form, this continuation is popped from the stack and entered.
- The tag of the data object is returned on top of the value stack.

- The components of the data object (if any) are returned in a frame pointed to by a new register, the *data frame pointer*.

So the code we would produce for `is_empty` would be like this[5]:

```
is_empty:       Take 1 1                -- One argument
                Push (Label "cont")     -- Continuation
                Enter (Arg 1)           -- Evaluate xs

cont:           Switch [ 1 -> [PushV (IntVConst 1), Return]
                         2 -> [PushV (IntVConst 0), Return]
                ]
```

The `Switch` instruction does a multi-way jump based on the top item on the value stack. In this example, both branches of the `case` expression just return a constant number.

In this example the components of the scrutinised list cell were not used. This is not always the case. Consider, for example, the `sum` function:

```
        sum xs = case xs of
                        <1>      -> 0
                        <2> y ys -> y + sum ys
```

`sum` computes the sum of the elements of a list. The new feature is that the expression `y + sum ys` uses the components, `y` and `ys`, of the list cell. As indicated earlier, these components are returned to the continuation in a frame pointed to by the data frame pointer, a new register. (Exercise: why cannot the ordinary frame pointer be used for this purpose?)

So far, every local variable (that is, supercombinator argument or `let(rec)`-bound variable) has a slot in the current frame which contains its closure, so it seems logical to extend the idea, and add further slots for `y` and `ys`. All we need to do is to move the closures out of the list cell frame, and into the current frame. Here, then, is the code for `sum`:

```
sum:    Take 3 1                -- One argument, two extra slots for y,ys
        Push (Label "cont")     -- Continuation for case
        Enter (Arg 1)           -- Evalute xs

cont:   Switch [
            1 -> [PushV (IntVConst 0), Return]
            2 -> [Move 2 (Data 1)
                  Move 3 (Data 2)
                  ...code to compute y + sum ys...
                 ]
        ]
```

The `Move` instructions use a new addressing mode `Data`, which addresses a closure in the frame pointed to by the data frame pointer. The two `Move` instructions copy `y` and `ys` from the list cell into the current frame (the one which contains `xs`).

[5]As usual we write the code with explicit labels for continuations, but in reality we would compile uses of the `Code` addressing mode so as to avoid generating fresh labels.

In summary, a `case` expression is compiled into five steps:

1. Push a continuation.
2. Enter the closure to be scrutinised. When it is evaluated, it will enter the continuation pushed in Step 1.
3. The continuation uses a `Switch` instruction to take a multi-way jump based on the tag, which is returned on top of the value stack.
4. Each branch of the `Switch` begins with `Move` instructions to copy the contents of the data object into the current frame. Since this copies the closure, we must be sure that all closures in data objects have the property that they can freely be copied (Section 4.5.6).
5. Each alternative then continues with the code for that alternative, compiled exactly as usual.

Finally, we can ask what code should be generated for the expression `Pack{`**tag**, **arity**`}`. Consider, for example, the expression

`Pack{1,2}` $e1$ $e2$

which builds a list cell. The minimalist approach is to treat `Pack{1,2}` as a supercombinator, and generate the following code[6]:

```
Push (...addressing mode for e2...)
Push (...addressing mode for e1...)
Enter (Label "Pack{1,2}")
```

The code for `Pack{1,2}` is very simple:

```
Pack{1,2}:      UpdateMarkers 2
                Take 2 2
                ReturnConstr 1
```

The first two instructions are just the same as for any other supercombinator. The `UpdateMarkers` instruction performs any necessary updates, and the `Take` instruction builds the frame containing the two components of the list cell, putting a pointer to it in the current frame pointer. Finally, a new instruction, `ReturnConstr`, enters the continuation, while pushing a tag of `1` onto the value stack, and copying the current frame pointer into the data frame pointer. Like `Return`, `ReturnConstr` needs to check for updates and perform them when necessary.

4.6.2 Transition rules and compilation schemes for data structures

Now that we have completed the outline, we can give the details of the transition rules and compilation schemes for the new constructs. The rule for `Switch` is as follows:

[6] In principle there are an infinite number of possible constructors, so it seems that we need an infinite family of similar code fragments for them in the code store. In practice this is easily avoided as will be seen when we write the detailed compilation schemes.

$$
(4.18)\quad
\begin{array}{|llllllll|}
\hline
 & [\texttt{Switch}\ [\ldots\ \mathbf{t}\ \texttt{->}\ \mathbf{i}\ \ldots]] & \mathbf{f} & \mathbf{f_d} & \mathbf{s} & \mathbf{t}:\mathbf{v} & \mathbf{d} & \mathbf{h}\ \ \mathbf{c} \\
\Longrightarrow & \mathbf{i} & \mathbf{f} & \mathbf{f_d} & \mathbf{s} & \mathbf{v} & \mathbf{d} & \mathbf{h}\ \ \mathbf{c} \\
\hline
\end{array}
$$

There are two rules for `ReturnConstr`, because it has to account for the possibility that an update is required. The first is straightforward, when there is no update to be done:

$$
(4.19)\quad
\begin{array}{|lllllllll|}
\hline
 & [\texttt{ReturnConstr}\ \mathbf{t}] & \mathbf{f} & \mathbf{f_d} & (\mathbf{i},\mathbf{f}'):\mathbf{s} & \mathbf{v} & \mathbf{d} & \mathbf{h} & \mathbf{c} \\
\Longrightarrow & \mathbf{i} & \mathbf{f}' & \mathbf{f} & \mathbf{s} & \mathbf{t}:\mathbf{v} & \mathbf{d} & \mathbf{h} & \mathbf{c} \\
\hline
\end{array}
$$

The second rule deals with updating, overwriting the closure to be updated with a code sequence containing only a `ReturnConstr` instruction, and the data frame pointer:

$$
(4.20)\quad
\begin{array}{|lllllllll|}
\hline
 & [\texttt{ReturnConstr}\ \mathbf{t}] & \mathbf{f} & \mathbf{f_d} & [] & \mathbf{v} & (\mathbf{f_u},\mathbf{x},\mathbf{s}):\mathbf{d} & \mathbf{h} & \mathbf{c} \\
\Longrightarrow & [\texttt{ReturnConstr}\ \mathbf{t}] & \mathbf{f} & \mathbf{f_d} & \mathbf{s} & \mathbf{v} & \mathbf{d} & \mathbf{h}' & \mathbf{c} \\
 & \multicolumn{8}{l}{\text{where } \mathbf{h}' = \mathbf{h}[\mathbf{f_u} : \langle \ldots, \mathbf{d_{x-1}}, ([\texttt{ReturnConstr}\ \mathbf{t}], \mathbf{f}), \mathbf{d_{x+1}}, \ldots \rangle]} \\
\hline
\end{array}
$$

The only changes to the compilation schemes are to add extra cases to the $\mathcal{R}$ scheme for constructors and for `case` expressions. The latter is structured by the use of an auxiliary scheme $\mathcal{E}$, which compiles a `case` alternative (Figure 4.6). Notice that constructors are compiled 'in-line' as they are encountered, which avoids the need for an infinite family of definitions to be added to the code store.

4.6.3 Trying it out

We can use the new machinery to implement lists and booleans, by using the following extra Core-language definitions:

```
cons = Pack{2,2}
nil  = Pack{1,0}

true  = Pack{2,0}
false = Pack{1,0}
if cond tbranch fbranch = case cond of
                            <1> -> fbranch
                            <2> -> tbranch
```

Notice that `if`, which previously had a special instruction and case in the compilation schemes, is now just a supercombinator definition like any other. Even so, it is often clearer to write programs using `if` rather than `case`, so you may want to leave the special case in your compiler; but now you can generate a `Switch` instruction rather than a `Cond` instruction. (The latter can vanish.)

Exercise 4.23. Implement the new instructions and compilation schemes.

Test your new implementation on the following program:

```
length xs = case xs of
                  <1>      -> 0
```

$$
\begin{array}{lcl}
\mathcal{R}[\![\texttt{Pack\{}\mathbf{t}\texttt{,}\mathbf{a}\texttt{\}}]\!]\ \rho\ \mathbf{d} & = & (\mathbf{d},\ [\texttt{UpdateMarkers}\ \mathbf{a},\ \texttt{Take}\ \mathbf{a}\ \mathbf{a},\ \texttt{ReturnConstr}\ \mathbf{t}]) \\
\\
\mathcal{R}[\![\texttt{case}\ \mathbf{e}\ \texttt{of}\ \mathbf{alt}_1\ \ldots\ \mathbf{alt_n}]\!]\ \rho\ \mathbf{d} & = & (\mathbf{d}',\ \texttt{Push}\ (\texttt{Code}\ [\texttt{Switch}\ [\mathbf{branch}_1\ \ldots\ \mathbf{branch_n}]]) : \mathbf{is_e}) \\
 & & \text{where } (\mathbf{d}_1, \mathbf{branch}_1) \ = \ \mathcal{E}[\![\mathbf{alt}_1]\!]\ \rho\ \mathbf{d} \\
 & & \qquad \ldots \\
 & & \qquad (\mathbf{d_n}, \mathbf{branch_n}) \ = \ \mathcal{E}[\![\mathbf{alt_n}]\!]\ \rho\ \mathbf{d} \\
 & & \qquad (\mathbf{d}', \mathbf{is_e}) \ = \ \mathcal{R}[\![\mathbf{e}]\!]\ \rho\ \mathbf{max}(\mathbf{d}_1, \ldots, \mathbf{d_n})
\end{array}
$$

$\mathcal{E}[\![\mathbf{alt}]\!]\ \rho\ \mathbf{d}$, where **alt** is a `case` alternative, is a pair $(\mathbf{d}', \mathbf{branch})$, where **branch** is the `Switch` branch compiled in environment ρ. The code assumes that the first **d** slots of the frame are occupied, and it uses slots $(\mathbf{d}+1 \ldots \mathbf{d}')$.

$$
\begin{array}{lcl}
\mathcal{E}[\![\texttt{<t>}\ \mathbf{x}_1 \ldots \mathbf{x_n}\ \texttt{->}\ \mathbf{body}]\!]\ \rho\ \mathbf{d} & = & (\mathbf{d}',\ \mathbf{t}\ \texttt{->}\ (\mathbf{is_{moves}} +\!\!+ \mathbf{is_{body}})) \\
 & & \text{where } \mathbf{is_{moves}} \ = \ [\texttt{Move}\ (\mathbf{d}+1)\ (\texttt{Data}\ 1), \\
 & & \qquad\qquad \ldots, \\
 & & \qquad\qquad \texttt{Move}\ (\mathbf{d}+\mathbf{n})\ (\texttt{Data}\ \mathbf{n})] \\
 & & \qquad (\mathbf{d}', \mathbf{is_{body}}) \ = \ \mathcal{R}[\![\mathbf{body}]\!]\ \rho'\ (\mathbf{d}+\mathbf{n}) \\
 & & \qquad \rho' \ = \ \rho[\mathbf{x}_1 \mapsto \texttt{Arg}\ (\mathbf{d}+1), \\
 & & \qquad\qquad \ldots, \\
 & & \qquad\qquad \mathbf{x_n} \mapsto \texttt{Arg}\ (\mathbf{d}+\mathbf{n})]
\end{array}
$$

Figure 4.6: Compilation schemes for `case` expressions

```
                  <2> p ps -> 1 + length ps

main = length (cons 1 (cons 2 nil))
```

A more interesting example, which will demonstrate whether your update code is working correctly, is this:

```
append xs ys = case xs of
                  <1>      -> ys
                  <2> p ps -> cons p (append ps ys)

main = let xs = append (cons 1 nil) (cons 2 nil)
       in
       length xs + length xs
```

Here `xs` is used twice, but the work of appending should only be done once.

Exercise 4.24. If the arity, **a**, of the constructor is zero, then $\mathcal{R}[\![\texttt{Pack\{}\mathbf{t}\texttt{,}\mathbf{a}\texttt{\}}]\!]$ will generate the code `[UpdateMarkers 0, Take 0 0, ReturnConstr` **t**`]`. Optimise the $\mathcal{R}$ scheme and `compileR` function to generate better code for this case (cf. Exercise 4.22).

4.6.4 Printing a list

The example programs suggested so far have all returned an integer, but it would be nice to be able to return and print a list instead.

The way we expressed this in the G-machine chapter was to add an extra component to the machine state to represent the *output*, together with an instruction `Print`, which appends a number to the end of the output. In our case, numbers are returned on the value stack, so `Print` consumes a number from the value stack and appends it to the output.

At present `compile` initialises the stack with the continuation `([],FrameNull)`, which has the effect of stopping the machine when it is entered. All we need to do is change this continuation to do the printing. This time, the continuation expects the value of the program to be a list, so it must do case analysis to decide how to proceed. If the list is empty, the machine should halt, so that branch can just have the empty code sequence. Otherwise, the head of the list should be evaluated and printed, and the tail then given the original continuation again. Here is the code:

```
topCont:        Switch [ 1 -> []
                         2 -> [ Move 1 (Data 1)         -- Head
                                Move 2 (Data 2)         -- Tail
                                Push (Label "headCont")
                                Enter (Arg 1)           -- Evaluate head
                              ]
                ]

headCont:       Print
                Push (Label "topCont")
                Enter (Arg 2)                           -- Do the same to tail
```

Notice that the `topCont` code needs a 2-slot frame for working storage, which `compile` had better provide for it. `compile` therefore initialises the stack with the continuation

```
(topCont, frame)
```

where `topCont` is the code sequence above, and `frame` is the address of a 2-slot frame allocated from the heap.

Exercise 4.25. Implement list printing as described. The only tiresome aspect is that you need to add an extra component to the machine state (again).

As usual, you can use `Push (Code ...)` instead of `Push (Label "headCont")`, and in fact you can do the same for `Push (Label "topCont")`, by using a little recursion!

Test your work on the following program:

```
between n m = if (n>m) nil (cons n (between (n+1) m))
main = between 1 4
```

Exercise 4.26. When running a program whose result is a list, it is nice to have the elements of the list printed as soon as they become available. With our present implementation, either we print every state (if we use `showFullResults`) or we print only the last state (using `showResults`). In the former case we get far too much output, while in the latter we get no output at all until the program terminates.

Modify `showResults` so that it prints the output as it is produced. The easiest way to do this is to compare the output component of successive pairs of states, and to print the last element when the output gets longer between one state and the next.

Another possible modification to `showResults` is to print a dot for each state (or ten states), to give a rough idea of how much work is done between each output step.

4.6.5 Using data structures directly†

One might ask why we cannot use the components of a data structure directly in the arms of a `Switch` instruction, by using `Data` addressing modes in instructions other than `Move`. The reason can be found in the `sum` example, which we repeat here:

```
sum xs = case xs of
                <1>      -> 0
                <2> y ys -> y + sum ys
```

Now, let us follow the code for `y + sum ys` a little further. This code must first evaluate `y`, which may take a lot of computation, certainly using the data frame pointer register. Hence, by the time it comes to evaluate `ys`, the data frame pointer will have been changed, so `ys` will no longer be accessible via the data frame pointer. By moving the contents of the list cell into the current frame, we enable them to be preserved across further evaluations.

Sometimes, no further evaluation is to be done, as in the `head` function:

```
head xs = case xs of
                <1>      -> error
                <2> y ys -> y
```

In this case, as an optimisation we could use `y` directly from the data frame; that is, the second branch of the `Switch` instruction would be simply [`Enter` (`Data` 1)].

Similarly, if a variable is not used at all in the branch of the `case` expression, there is no need to move it into the current frame.

4.7 Mark 6: Constant applicative forms and the code store†

As we mentioned earlier (Section 4.2.3), our decision to represent the code store as an association list of names and code sequences means that CAFs do not get updated. Instead, their code is executed each time they are called, which will perhaps duplicate work. We would like to avoid this extra work, but the solution for the TIM is not quite as easy as that for our earlier implementations.

In the case of the template instantiation machine and the G-machine, the solution was to allocate a node in the heap to represent each supercombinator. When a CAF is called, the root of the redex is the supercombinator node itself, and so the node is updated with the result of the reduction (that is, an instance of the right-hand side of the supercombinator definition). Any subsequent use of the supercombinator will see this updated node instead of the original supercombinator. The trouble is that the TIM does not have heap nodes at all; what corresponds to a node is a closure within a frame. So what we have to do is to allocate in the initial heap a single giant frame, the *global frame*, which contains a closure for each supercombinator.

The code store is now represented by the address, $\mathbf{f_G}$, of the global frame, together with an association list, $\mathbf{g}$, mapping supercombinator names to their offset in the frame. The `Label` addressing mode uses this association list to find the offset, and then fetches the closure for the supercombinator from the global frame. The new transition rule for `Push Label` formalises these ideas:

(4.21)

$$\boxed{\begin{array}{llllll} & \texttt{Push (Label } \mathbf{l}\texttt{)} : \mathbf{i} & \mathbf{f} & \mathbf{s} & \mathbf{h}[\mathbf{f_G} : \langle(\mathbf{i}_1, \mathbf{f}_1), \ldots, (\mathbf{i_n}, \mathbf{f_n})\rangle] & (\mathbf{f_G}, \mathbf{g}[\mathbf{l} : \mathbf{k}]) \\ \Longrightarrow & \mathbf{i} & \mathbf{f} & (\mathbf{i_k}, \mathbf{f_k}) : \mathbf{s} & \mathbf{h} & (\mathbf{f_G}, \mathbf{g}) \end{array}}$$

The rule for `Enter Label` follows directly from the `Push`/`Enter` relationship. Each closure in the global frame is a self-updating closure, as described in the context of `let(rec)`-bound variables in Section 4.5.5. Just as for `let(rec)`-bound variables, when pushing a supercombinator on the stack we should use a (non-updating) indirection (Section 4.5.5).

4.7.1 Implementing CAFs

Here is what needs to be done to add proper updating for CAFs to a Mark 4 or Mark 5 TIM.

- The code store component of the machine state now contains the address of the global frame and an association between supercombinator names and frame offsets:

  ```
  > codeStore == (addr, assoc name num)
  ```

  The `showSCDefns` function must be altered to take account of this change.

- The function `amToClosure` must take different action for a `Label` addressing mode, as described above.

- The initial environment, `initial_env`, computed in the `compile` function, must be altered to generate an indirection addressing mode for each supercombinator.

- The last modification involves most work. We need to alter the `compile` function to build the initial heap, just as we did in the `compile` function of the template instantiation machine and the G-machine.

The last of these items needs a little more discussion. Instead of starting with an empty heap, `compile` now needs to build an initial heap, using an auxiliary function `allocateInitialHeap`. `allocateInitialHeap` is passed the `compiled_code` from the `compile` function. It allocates a single big frame containing a closure for each element of `compiled_code`, and returns the initial heap and the `codeStore`:

```
> allocateInitialHeap :: [(name, [instruction])] -> (timHeap, codeStore)
> allocateInitialHeap compiled_code
>  = (heap, (global_frame_addr, offsets))
>    where
>    indexed_code = zip2 [1..] compiled_code
>    offsets = [(name, offset) | (offset, (name, code)) <- indexed_code]
>    closures = [(PushMarker offset : code, global_frame_addr) |
>                         (offset, (name, code)) <- indexed_code]
>    (heap, global_frame_addr) = fAlloc hInitial closures
```

`allocateInitialHeap` works as follows. First the `compiled_code` list is indexed, by pairing each element with a frame offset, starting at 1. Now this list is separately processed to produce `offsets`, the mapping from supercombinator names to addresses, and `closures`, the list of closures to be placed in the global frame. Finally, the global frame is allocated, and the resulting heap is returned together with the code store.

Notice that `global_frame_addr` is used in constructing `closures`; the frame pointer of each supercombinator closure is the global frame pointer itself, so that the `PushMarker` instruction pushes an update frame referring to the global frame.

Exercise 4.27. Make the required modifications to `showSCDefns`, `compileA`, `amToClosure` and `compile`. Test whether updating of CAFs does in fact take place.

Exercise 4.28. The `PushMarker` instruction added inside `allocateInitialHeap` is only required for CAFs, and is a waste of time for supercombinators with one or more arguments. Modify `allocateInitialHeap` to plant the `PushMarker` instruction only for CAFs. (Hint: you can identify non-CAFs by the fact that their code begins with a `Take` **n** instruction, where $\mathbf{n} > 0$.) Measure the improvement.

Exercise 4.29. An indirection addressing mode is only required for CAFs, and not for non-CAF supercombinators. Modify the construction of `initial_env` to take advantage of this fact.

4.7.2 Modelling the code store more faithfully

There is something a little odd about our handling of `Label`s so far. It is this: the names of supercombinators get looked up in the environment at compile-time (to map them to a `Label` addressing mode), and then again at run-time (to map them to an offset in the global frame[7]). This is hardly realistic: in a real compiler, names will be looked up at compile-time, but will be linked to a hard machine address before run-time, so no run-time lookups take place.

We can model this by changing the `Label` constructor to take two arguments instead of one, thus:

```
timAMode ::= Label name num
           | ...as before...
```

[7]We are assuming that we have implemented the changes suggested in the previous section for CAFs, but this section applies also to the pre-CAF versions of the machine.

The `name` field records the name of the supercombinator as before, but now the `num` says what offset to use in the global frame. Just as in the `NSupercomb` constructor of the template machine, the `name` field is only there for documentation and debugging purposes. The code store component now becomes simply the address of the global frame, as you can see from the revised rule for `Push Label`:

$$(4.22)\quad \boxed{\begin{array}{lrlrll} & \texttt{Push (Label l k)} : \mathbf{i} & \mathbf{f} & \mathbf{s} & \mathbf{h}[\mathbf{g} : \langle(\mathbf{i}_1,\mathbf{f}_1),\dots,(\mathbf{i_n},\mathbf{f_n})\rangle] & \mathbf{g} \\ \Longrightarrow & \mathbf{i} & \mathbf{f} & (\mathbf{i_k},\mathbf{f_k}) : \mathbf{s} & \mathbf{h} & \mathbf{g} \end{array}}$$

The rule for `Enter` follows from the `Push`/`Enter` relationship.

Exercise 4.30. Implement this idea. To do this:

- Change the `timAMode` type as described.
- Change the `codeStore` type to consist only of a frame pointer.
- Change the `compile` function so that it generates the correct initial state for the machine. In particular, it must generate an `initial_env` with the right `Label` addressing modes.
- Adjust the `show` functions to account for these changes.

4.8 Summary

The final TIM compilation schemes are summarised in Figures 4.7 and 4.8. The obvious question is 'is the TIM better or worse than the G-machine?'; it is a hard one to answer. Our prototypes are very useful for exploring design choices, but really no good at all for making serious performance comparisons. How can one establish, for example, the relative costs of a `Take` instruction compared with a G-machine `Mkap`? About the only really comparable measure we have available is the heap consumption of the two.

Still, it can be very illuminating to explore another evaluation model, as we have done in this chapter, because it suggests other design avenues which combine aspects of the TIM with those of the G-machine. One attempt to do so is the Spineless Tagless G-machine [Peyton Jones and Salkild 1989, Peyton Jones 1991], which adopts the spinelessness and update mechanism of TIM, but whose stack consists of pointers to heap objects (like the G-machine) rather than code-frame pairs (as in TIM).

$\mathcal{SC}[\![\mathbf{def}]\!]\ \rho$ is the TIM code for the supercombinator definition **def** compiled in environment ρ.

$$\mathcal{SC}[\![\mathbf{f}\ \mathbf{x}_1\ \ldots\ \mathbf{x_n}\ =\ \mathbf{e}]\!]\ \rho\ =\ \begin{array}{l}\texttt{UpdateMarkers}\ \mathbf{n}\ :\ \texttt{Take}\ \mathbf{d}'\ \mathbf{n}\ :\ \mathbf{is} \\ \text{where } (\mathbf{d}', \mathbf{is}) = \mathcal{R}[\![\mathbf{e}]\!]\ \rho[\mathbf{x}_1 \mapsto \texttt{Arg}\ 1, \ldots, \mathbf{x_n} \mapsto \texttt{Arg}\ \mathbf{n}]\ \mathbf{n}\end{array}$$

$\mathcal{R}[\![\mathbf{e}]\!]\ \rho\ \mathbf{d}$ is a pair $(\mathbf{d}', \mathbf{is})$, where **is** is TIM code which applies the value of the expression **e** in environment ρ to the arguments on the stack. The code **is** assumes that the first **d** slots of the frame are occupied, and it uses slots $(\mathbf{d}+1 \ldots \mathbf{d}')$.

$$\begin{array}{rcl}
\mathcal{R}[\![\mathbf{e}]\!]\ \rho\ \mathbf{d} & = & \mathcal{B}[\![\mathbf{e}]\!]\ \rho\ \mathbf{d}\ [\texttt{Return}] \\
 & & \text{where } \mathbf{e} \text{ is an integer or arithmetic expression} \\[1em]
\mathcal{R}[\![\mathbf{a}]\!]\ \rho\ \mathbf{d} & = & (\mathbf{d},\ [\texttt{Enter}\ (\mathcal{A}[\![\mathbf{a}]\!]\ \rho)]) \\
 & & \text{where } \mathbf{a} \text{ is a supercombinator or local variable} \\[1em]
\mathcal{R}[\![\mathbf{e}\ \mathbf{a}]\!]\ \rho\ \mathbf{d} & = & (\mathbf{d}_1,\ \texttt{Push}\ (\mathcal{A}[\![\mathbf{a}]\!]\ \rho)\ :\ \mathbf{is}) \\
 & & \text{where}\ \ \mathbf{a} \text{ is a supercombinator, local variable, or integer} \\
 & & \qquad\quad (\mathbf{d}_1, \mathbf{is})\ =\ \mathcal{R}[\![\mathbf{e}]\!]\ \rho\ \mathbf{d} \\[1em]
\mathcal{R}[\![\mathbf{e_{fun}}\ \mathbf{e_{arg}}]\!]\ \rho\ \mathbf{d} & = & (\mathbf{d}_2,\ \texttt{Move}\ (\mathbf{d}+1)\ \mathbf{am_{arg}} : \texttt{Push}\ \mathcal{I}[\![\mathbf{d}+1]\!]\ : \mathbf{is_{fun}}) \\
 & & \text{where}\ \ (\mathbf{d}_1, \mathbf{am_{arg}})\ =\ \mathcal{U}[\![\mathbf{e_{arg}}]\!]\ (\mathbf{d}+1)\ \rho\ (\mathbf{d}+1) \\
 & & \qquad\quad (\mathbf{d}_2, \mathbf{is_{fun}})\ =\ \mathcal{R}[\![\mathbf{e_{fun}}]\!]\ \rho\ \mathbf{d}_1
\end{array}$$

$$\begin{array}{l}
\mathcal{R}[\![\texttt{let}\ \mathbf{x}_1\texttt{=}\mathbf{e}_1\texttt{;}\ \ldots\texttt{;}\ \mathbf{x_n}\texttt{=}\mathbf{e_n}\ \texttt{in}\ \mathbf{e}]\!]\ \rho\ \mathbf{d} \\
\qquad = (\mathbf{d}',\ [\texttt{Move}\ (\mathbf{d}+1)\ \mathbf{am}_1,\ \ldots, \texttt{Move}\ (\mathbf{d}+\mathbf{n})\ \mathbf{am_n}]\ +\!\!+\ \mathbf{is}) \\
\qquad\quad \text{where}\ \begin{array}[t]{lcl}
(\mathbf{d}_1, \mathbf{am}_1) & = & \mathcal{U}[\![\mathbf{e}_1]\!]\ (\mathbf{d}+1)\ \rho\ (\mathbf{d}+\mathbf{n}) \\
(\mathbf{d}_2, \mathbf{am}_2) & = & \mathcal{U}[\![\mathbf{e}_2]\!]\ (\mathbf{d}+2)\ \rho\ \mathbf{d}_1 \\
\ldots & & \\
(\mathbf{d_n}, \mathbf{am_n}) & = & \mathcal{U}[\![\mathbf{e_n}]\!]\ (\mathbf{d}+\mathbf{n})\ \rho\ \mathbf{d_{n-1}} \\
\rho' & = & \rho[\mathbf{x}_1 \mapsto \mathcal{I}[\![\mathbf{d}+1]\!], \ldots, \mathbf{x_n} \mapsto \mathcal{I}[\![\mathbf{d}+\mathbf{n}]\!]\,] \\
(\mathbf{d}', \mathbf{is}) & = & \mathcal{R}[\![\mathbf{e}]\!]\ \rho'\ \mathbf{d_n}
\end{array}
\end{array}$$

The `letrec` case is similar, except that ρ' is passed to the calls to $\mathcal{U}[\![]\!]$ instead of ρ.

$$\mathcal{R}[\![\texttt{Pack\{}\mathbf{t}\texttt{,}\mathbf{a}\texttt{\}}]\!]\ \rho\ \mathbf{d}\ =\ (\mathbf{d},\ [\texttt{UpdateMarkers}\ \mathbf{a},\ \texttt{Take}\ \mathbf{a}\ \mathbf{a},\ \texttt{ReturnConstr}\ \mathbf{t}])$$

$$\begin{array}{l}
\mathcal{R}[\![\texttt{case}\ \mathbf{e}\ \texttt{of}\ \mathbf{alt}_1\ \ldots\ \mathbf{alt_n}]\!]\ \rho\ \mathbf{d} \\
\qquad = (\mathbf{d}',\ \texttt{Push}\ (\texttt{Code}\ [\texttt{Switch}\ [\mathbf{branch}_1\ \ldots\ \mathbf{branch_n}]])\ :\ \mathbf{is_e}) \\
\qquad\quad \text{where}\ \begin{array}[t]{lcl}
(\mathbf{d}_1, \mathbf{branch}_1) & = & \mathcal{E}[\![\mathbf{alt}_1]\!]\ \rho\ \mathbf{d} \\
\ldots & & \\
(\mathbf{d_n}, \mathbf{branch_n}) & = & \mathcal{E}[\![\mathbf{alt_n}]\!]\ \rho\ \mathbf{d} \\
(\mathbf{d}', \mathbf{is_e}) & = & \mathcal{R}[\![\mathbf{e}]\!]\ \rho\ \mathbf{max}(\mathbf{d}_1, \ldots, \mathbf{d_n})
\end{array}
\end{array}$$

Figure 4.7: Final TIM compilation schemes (part 1)

$\mathcal{E}[\![\mathbf{alt}]\!]\ \rho\ \mathbf{d}$, where **alt** is a `case` alternative, is a pair $(\mathbf{d}', \mathbf{branch})$, where **branch** is the `Switch` branch compiled in environment ρ. The code assumes that the first **d** slots of the frame are occupied, and it uses slots $(\mathbf{d}+1 \ldots \mathbf{d}')$.

$$
\begin{array}{lcl}
\mathcal{E}[\![\texttt{<t>}\ \mathbf{x}_1 \ldots \mathbf{x_n}\ \texttt{->}\ \mathbf{body}]\!]\ \rho\ \mathbf{d} & = & (\mathbf{d}',\ \mathbf{t}\ \texttt{->}\ (\mathbf{is_{moves}} +\!\!+\ \mathbf{is_{body}})) \\
 & & \quad \text{where } \mathbf{is_{moves}} = [\texttt{Move}\ (\mathbf{d}+1)\ (\texttt{Data}\ 1), \\
 & & \qquad\qquad\qquad\qquad \ldots, \\
 & & \qquad\qquad\qquad\qquad \texttt{Move}\ (\mathbf{d}+\mathbf{n})\ (\texttt{Data}\ \mathbf{n})] \\
 & & \qquad\quad (\mathbf{d}', \mathbf{is_{body}}) = \mathcal{R}[\![\mathbf{e}]\!]\ \rho'\ (\mathbf{d}+\mathbf{n}) \\
 & & \qquad\quad \rho' = \rho[\mathbf{x}_1 \mapsto \texttt{Arg}\ (\mathbf{d}+1), \\
 & & \qquad\qquad\qquad \ldots, \\
 & & \qquad\qquad\qquad \mathbf{x_n} \mapsto \texttt{Arg}\ (\mathbf{d}+\mathbf{n})]
\end{array}
$$

$\mathcal{U}[\![\mathbf{e}]\!]\ \mathbf{u}\ \rho\ \mathbf{d}$ is a pair $(\mathbf{d}', \mathbf{am})$, where **am** is a TIM addressing mode for expression **e** in environment ρ. If the closure addressed by **am** is entered, it will update slot **u** of the current frame with its normal form. The code assumes that the first **d** slots of the frame are occupied, and it uses slots $(\mathbf{d}+1 \ldots \mathbf{d}')$.

$$
\begin{array}{lcll}
\mathcal{U}[\![\mathbf{n}]\!]\ \mathbf{u}\ \rho\ \mathbf{d} & = & (\mathbf{d},\ \texttt{IntConst}\ \mathbf{n}) & \text{where } \mathbf{n} \text{ is an integer constant} \\
\mathcal{U}[\![\mathbf{e}]\!]\ \mathbf{u}\ \rho\ \mathbf{d} & = & (\mathbf{d}',\ \texttt{Code}\ (\texttt{PushMarker}\ \mathbf{u} : \mathbf{is})) & \text{otherwise} \\
 & & \text{where } (\mathbf{d}', \mathbf{is}) = \mathcal{R}[\![\mathbf{e}]\!]\ \rho\ \mathbf{d} &
\end{array}
$$

$\mathcal{A}[\![\mathbf{e}]\!]\ \rho$ is a TIM addressing mode for expression **e** in environment ρ.

$$
\begin{array}{lcll}
\mathcal{A}[\![\mathbf{n}]\!]\ \rho & = & \texttt{IntConst}\ \mathbf{n} & \text{where } \mathbf{n} \text{ is a number} \\
\mathcal{A}[\![\mathbf{x}]\!]\ \rho & = & \rho\ \mathbf{x} & \text{where } \mathbf{x} \text{ is bound by } \rho
\end{array}
$$

$\mathcal{I}[\![\mathbf{d}]\!]$ is an indirection addressing mode for frame offset **d**

$$\mathcal{I}[\![\mathbf{d}]\!] = \texttt{Code}\ [\texttt{Enter}\ (\texttt{Arg}\ \mathbf{d})]$$

$\mathcal{B}[\![\mathbf{e}]\!]\ \rho\ \mathbf{d}\ \mathbf{cont}$ is a pair $(\mathbf{d}', \mathbf{is})$, where **is** is TIM code which evaluates **e** in environment ρ, putting its value (which should be an integer) on top of the value stack, and continuing with the code sequence **cont**. The code assumes that the first **d** slots of the frame are occupied, and it uses slots $(\mathbf{d}+1 \ldots \mathbf{d}')$.

$$
\begin{array}{lcl}
\mathcal{B}[\![\mathbf{e}_1\ \texttt{+}\ \mathbf{e}_2]\!]\ \rho\ \mathbf{d}\ \mathbf{cont} & = & \mathcal{B}[\![\mathbf{e}_2]\!]\ \rho\ \mathbf{d}_1\ \mathbf{is}_1 \\
 & & \text{where } (\mathbf{d}_1, \mathbf{is}_1) = \mathcal{B}[\![\mathbf{e}_1]\!]\ \rho\ \mathbf{d}\ (\texttt{Op Add}\ :\ \mathbf{cont})
\end{array}
$$

... and similar rules for other arithmetic primitives

$$
\begin{array}{lcll}
\mathcal{B}[\![\mathbf{n}]\!]\ \rho\ \mathbf{d}\ \mathbf{cont} & = & (\mathbf{d}, \texttt{PushV}\ (\texttt{IntVConst}\ \mathbf{n})\ :\ \mathbf{cont}) & \text{where } \mathbf{n} \text{ is a number} \\
\mathcal{B}[\![\mathbf{e}]\!]\ \rho\ \mathbf{d}\ \mathbf{cont} & = & (\mathbf{d}', \texttt{Push}\ (\texttt{Code}\ \mathbf{cont})\ :\ \mathbf{is}) & \text{otherwise} \\
 & & \text{where } (\mathbf{d}', \mathbf{is}) = \mathcal{R}[\![\mathbf{e}]\!]\ \rho\ \mathbf{d} &
\end{array}
$$

Figure 4.8: Final TIM compilation schemes (part 2)

Chapter 5

A Parallel G-machine

5.1 Introduction

In this chapter we develop an abstract machine and compiler for a parallel G-machine. It is based on a simplified version of the parallel G-machine developed as part of ESPRIT project 415; interested readers are referred to [Kingdon *et al* 1991] for an easily accessible account. A general introduction to parallel graph reduction can be found in [Peyton Jones 1989].

5.1.1 Parallel functional programming

Writing parallel imperative programs is hard. Amongst the reasons for this are the following:

- The programmer has to conceive of a *parallel algorithm* which meets the specification of the problem.
- The algorithm must be translated into the programming language constructs provided by the language. This is likely to entail: identification of *concurrent tasks*, defining the interfaces between tasks to allow them to *synchronise* and *communicate*. *Shared data* may need to be especially protected, to prevent more than one task accessing a variable at once.
- The programmer may be responsible for assigning tasks to processors, ensuring that tasks that need to communicate with one another are assigned to processors that are physically connected.
- Finally, in systems with no programmer control over the *scheduling policy*, the programmer must prove that the collection of concurrent tasks will execute correctly under all possible *interleavings* of task operations.

In contrast, when programming in a functional language, only the first of these points applies. Consider the following (contrived) example program, `psum n`, which calculates the sum of the numbers $1 \ldots$ `n`.

```
psum n = dsum 1 n;
dsum lo hi = let mid = (lo+hi)/2 in
             if (hi==lo) hi ((dsum lo mid)+(dsum (mid+1) hi))
```

The dsum function works by dividing the problem into two, roughly equal, parts. It then combines the two results together to generate the answer. This is a classic *divide-and-conquer algorithm*.

Notice that neither of the functions dsum or psum includes any mention of parallel primitives in the language; so why is psum a parallel algorithm? For comparison, we can write a sequential algorithm: ssum.

```
ssum n = if (n==1) 1 (n + ssum (n-1))
```

This function is a sequential algorithm because its *data dependencies* are inherently sequential. In the ssum example, what we mean by this is that the addition in ssum n can only take place once ssum (n-1) has been evaluated. This in turn can only take place once we have evaluated ssum (n-2), and so on. We may summarise this distinction as:

> *A function implements a parallel algorithm whenever it permits the concurrent evaluation of two or more sub-expressions of its body.*

As with any other programming language, a parallel algorithm is essential. Notice, however, the contrasts with parallel imperative programming:

- No new language constructs are required to express parallelism, synchronisation or communication. The concurrency is implicit, and new tasks are created dynamically to be executed by the machine whenever it has spare capacity.
- No special measures are taken to protect data shared between concurrent tasks. For example, mid is safely shared between the two concurrent tasks in dsum.
- We need no new proof techniques to reason about the parallel programs, as all of the techniques we use for sequentially executed functional programs still work. We also note that *deadlock* can only arise as a result of a self-dependency, such as

  ```
  letrec a = a+1 in a
  ```

  Expressions depending on their own value are meaningless, and their non-termination is the same behaviour as observed in the sequential implementation.
- The results of a program are determinate. It is not possible for the scheduling algorithm to cause answers to differ between two runs of the same program.

In summary, we suggest that these features allow us to express parallel algorithms conveniently, without having to solve a large number of low-level problems. Perhaps we can characterise this as:

> *A parallel imperative program specifies in detail* resource allocation decisions *which a parallel functional program does not even mention.*

This means that the machine will have to be able to make resource allocation decisions automatically. We will pay, with a loss of execution efficiency, whenever these decisions are not optimal.

Annotations

The high level of abstraction offered by functional languages places heavy demands on the compile-time and run-time resource allocators. Rather than leave *all* resource allocation decisions to the system, we will introduce an *annotation*, `par`, which initiates a new parallel thread. The annotation is a meaning-preserving decoration of the program text; in the case of `par` it has the following syntax and meaning:

$$\texttt{par}\ \mathbf{E}_1\ \mathbf{E}_2 = \mathbf{E}_1\ \mathbf{E}_2$$

That is: `par` is a synonym for application. Where it differs is that we intend that the expression $\mathbf{E}_2$ should be evaluated by a concurrent task. As an example we will now rewrite the `dsum` function using this annotation:

```
dsum lo hi = let mid = (lo+hi)/2 in
             let add x y = x+y
             if (lo==hi) hi (par (add (dsum lo mid))
                                 (dsum (mid+1) hi))
```

We see that `par` causes the second argument to `+` to be evaluated in parallel. The `par` annotation can be inserted by the programmer or, in principle, by a clever compiler. Such cleverness is, however, beyond the scope of this book, so we will assume that the `par`s have already been inserted.

5.1.2 Parallel graph reduction

We have seen in this book that graph reduction is a useful implementation technique for sequential machines; it will be no surprise that it is also suited to the implementation of parallel machines. In fact it has a number of benefits:

- There is no sequential concept of program counter; graph reduction is decentralised and distributed.
- Reductions may take place concurrently in many places in the graph, and the relative order in which they are scheduled cannot affect the result.
- All communication and synchronisation takes place via the graph.

A parallel model

A *task* is a sequential computation whose purpose is to reduce a particular sub-graph to WHNF. At any moment there may be many tasks that are able to run; this collection of sparked tasks is called the *spark pool*. A processor in search of work fetches a new task from the spark pool and executes it. A task can be thought of as a *virtual processor*.

Initially, there is only one task, whose job is to evaluate the whole program. During its execution, it will hopefully create new tasks to evaluate expressions that the main task will later need. These are placed in the spark pool, so that another processor can pick up the task if it has run out of work. We call the act of placing a task into the spark pool *sparking* a child task.

It is useful to consider the interaction between parent and child tasks. In the evaluate-and-die model of task management there is a lock bit on each graph node. When the bit is on, a task is executing which will evaluate the node; otherwise the bit is off. When a parent task requires the value of a sub-graph for which a task has been sparked, there are three cases to consider:

- the child task has not started yet,
- the child task has started, but not stopped yet, or
- the child task has completed.

In the first case, the parent task can evaluate the graph just as if the child task was not there. Of course this sets the lock bit, so that when an attempt is made to execute the child task it is immediately discarded. The interesting case is the second one, in which both parent and child are executing. When this is the situation, the parent task must wait for the child task to complete before proceeding. We say that the child task is *blocking* the parent task. In the third case, the node is now in WHNF and unlocked, so the parent task will not take very long to fetch the value of the graph.

The advantage of the evaluate-and-die model is that blocking only occurs when the parent and child actually collide. In all other cases, the parent's execution continues unhindered. Notice that the blocking mechanism is the *only* form of inter-task communication and synchronisation. Once a piece of graph has been evaluated to WHNF any number of tasks can simultaneously inspect it without contention.

An example

We begin with a sample execution of the simplest parallel program:

```
main = par I (I 3)
```

As we will shortly see, the two reductions of the identity function can take place in parallel.

A good compiler will produce the following code for the `main` supercombinator[1]:

```
e1 ++ par
where e1  = [Pushint 3, Pushglobal "I", Mkap, Push 0]
      par = [Par, Pushglobal "I", Mkap, Update 0, Pop 0, Unwind]
```

Initially, there is only one task executing; it is constructing the sub-expression `(I 3)`. After executing the code sequence `e1`, the machine will be in the state shown in diagram (a) of Figure 5.1.

After executing `e1` the machine encounters a `Par` instruction. This causes it to create a new task to evaluate the node pointed to from the top of the stack. We will refer to the new task as

[1]Your compiler might not produce this code if it is not sophisticated enough.

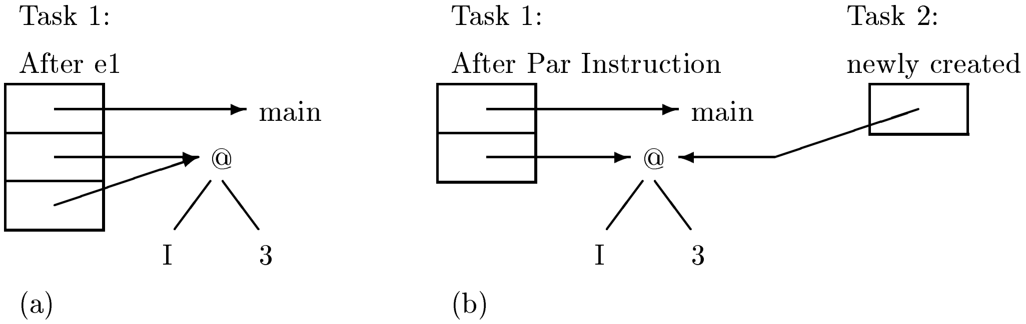


Figure 5.1: State after executing `e1` code sequence and `Par`

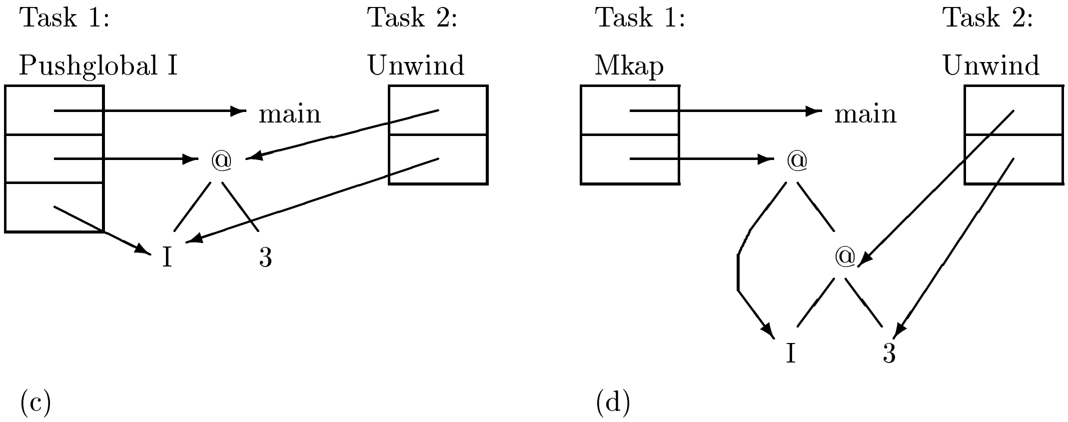


Figure 5.2: State after `Task 1` executes `[Pushglobal I, Mkap]`

`Task 2`; the original task will be labelled `Task 1`. This situation is illustrated in diagram (b) of Figure 5.1.

In diagram (c) (Figure 5.2), we see `Task 1` continuing with its evaluation; it is performing a `Pushglobal I` instruction. The newly created task – `Task 2` – starts with the code sequence: `[Unwind]`; it therefore starts to unwind the graph it has been assigned to evaluate. Diagram (d) shows `Task 1` completing the instantiation of the body of `main`, and `Task 2` completing its unwinding.

Its body instantiated, `Task 1` overwrites the redex node, which is `main`. `Task 2` performs a `Push 0` instruction, this being the first instruction in the code for the `I` supercombinator. This is shown in diagram (e) (Figure 5.3). In diagram (f) we see `Task 1` commence unwinding its spine, whilst `Task 2` performs its updating.

In Figure 5.4, we see `Task 2` run to completion. The remainder of the execution of `Task 1` is the same as the sequential G-machine, so we omit it.

This concludes a brief overview of the parallel G-machine's execution with concurrent tasks. We now provide a minimal parallel G-machine.

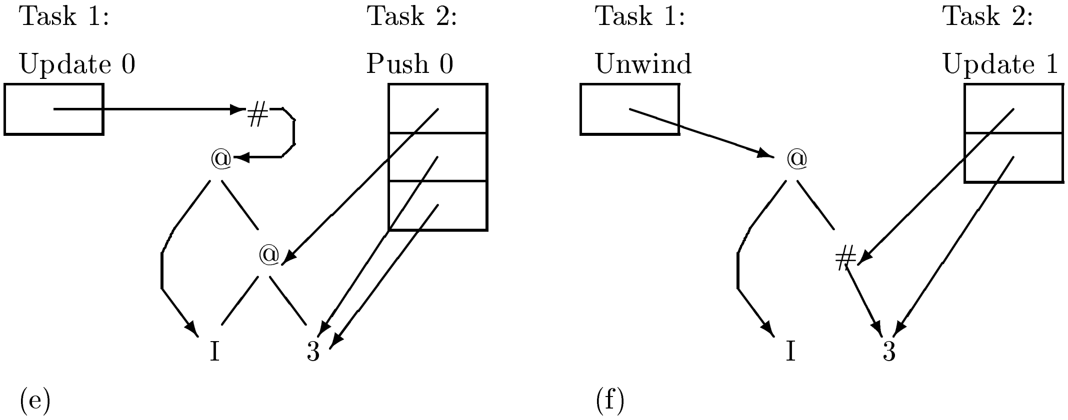


Figure 5.3: State after `Task 1` executes `[Update 0, Unwind]`

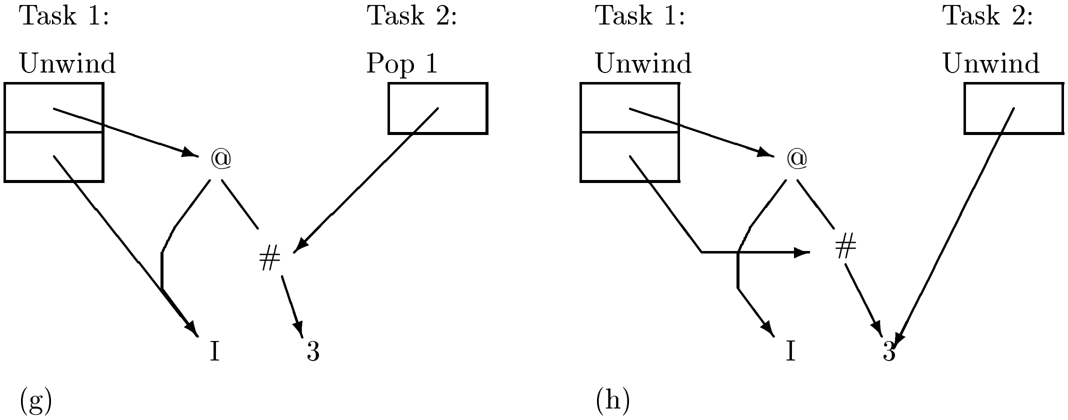


Figure 5.4: State after `Task 1` executes `[Unwind, Unwind]`

5.2 Mark 1: A minimal parallel G-machine

The first machine we present can be based on any of the G-machines from Chapter 3, except the Mark 1; we need this restriction to ensure that updating is done. To this basic machine we need to add the machinery for parallelism. We make the following basic assumptions.

1. There is a shared global graph, which all processors can access.
2. There are an infinite number of processors. This means that there is always a processor available to execute a task.
3. There is no locking of graph nodes. This means that it is possible that different tasks will re-evaluate the same expression.

5.2.1 Data type definitions

We will be making considerable use of state access functions in this chapter. Although they are not particularly interesting, for completeness they are included in this section.

In a parallel machine the state pgmState is split into two components: a global component, pgmGlobalState; and a local component, pgmLocalState. The pgmLocalState contains the processors that will execute the program. The pgmGlobalState contains global data structures – of which the heap is the most frequently used – that are manipulated by the processors.

```
> pgmState == (pgmGlobalState,      || Current global state
>              [pgmLocalState])     || Current states of processors
```

The global state component

To accommodate all of the possible machines you might use as a basis for your parallel implementation, the global state consists of five components: gmOutput, which is the output printed as the answer to the evaluation of the program; gmHeap, which is the heap; gmGlobals, which is used to point to the unique node in the heap for each supercombinator; gmSparks, which is the task pool and will be used to hold tasks before we begin to execute them; and gmStats, which is a global clock.

```
> pgmGlobalState == (gmOutput,          || output stream
>                    gmHeap,            || Heap of nodes
>                    gmGlobals,         || Global addresses in heap
>                    gmSparks,          || Sparked task pool
>                    gmStats)           || Statistics
```

We consider each of these components in turn.

- The gmOutput component was introduced in the Mark 6 G-machine (see Section 3.8). It is used to accumulate the result of executing programs that can generate structured data, and is simply a string.

  ```
  > gmOutput == [char]
  ```

  The function to get the gmOutput from a pgmState is pgmGetOutput is:

  ```
  > pgmGetOutput :: pgmState -> gmOutput
  > pgmGetOutput ((o, heap, globals, sparks, stats), locals) = o
  ```

- The heap data structure is the same as we used for the sequential G-machine.

  ```
  > gmHeap == heap node
  ```

  To get the heap from a pgmState we use pgmGetHeap.

  ```
  > pgmGetHeap :: pgmState -> gmHeap
  > pgmGetHeap ((o, heap, globals, sparks, stats), locals) = heap
  ```

- The addresses of global nodes in the heap are stored in gmGlobals; this too is the same structure we used in the sequential G-machine.

  ```
  > gmGlobals == assoc name addr
  ```

  Obtaining the gmGlobals from a pgmState is performed using the pgmGetGlobals function.

  ```
  > pgmGetGlobals :: pgmState -> gmGlobals
  > pgmGetGlobals ((o, heap, globals, sparks, stats), locals) = globals
  ```

- The spark pool is represented by the gmSparks component. It holds the addresses of nodes in the graph which have been marked by the par annotation as needing to be evaluated concurrently.

  ```
  > gmSparks == [addr]
  ```

  Access to this component is achieved by using the function pgmGetSparks.

  ```
  > pgmGetSparks :: pgmState -> gmSparks
  > pgmGetSparks ((o, heap, globals, sparks, stats), locals) = sparks
  ```

- Finally, in the parallel G-machine we will hold the accumulated statistics in the global component. It is represented as a list of numbers; these detail how long each task in the machine ran for, before it completed.

  ```
  > gmStats == [num]
  ```

  Access to this component is accomplished using pgmGetStats.

  ```
  > pgmGetStats :: pgmState -> gmStats
  > pgmGetStats ((o, heap, globals, sparks, stats), locals) = stats
  ```

The local state component

The local component of the parallel G-machine consists of a list of processors; a processor is represented as a task. Again, to make the parallel machine capable of executing with any G-machine as its basis, we make the state of each processor a 5-tuple:

```
> pgmLocalState == (gmCode,        || Instruction stream
>                   gmStack,       || Pointer stack
>                   gmDump,        || Stack of dump items
>                   gmVStack,      || Value stack
>                   gmClock)       || Number of ticks the task
>                                  ||          has been active
```

We now consider each component in turn.

- The code sequence is simply a list of instructions.

  ```
  > gmCode == [instruction]
  ```

- As in the sequential G-machine, the stack is a list of addresses in heap.

  ```
  > gmStack == [addr]
  ```

- If you are using a Mark 4 G-machine (or any later mark) as a basis for your parallel implementation, then a dump is needed. This is used as a stack of dump items, each of type gmDumpItem.

  ```
  > gmDump     == [gmDumpItem]
  > gmDumpItem == (gmCode, gmStack)
  ```

- If you have used the Mark 7 G-machine as the basis of your implementation you will need a V-stack for each processor.

  ```
  > gmVStack == [num]
  ```

- We also provide each processor with a clock. This records how many instructions the task has executed:

  ```
  > gmClock == num
  ```

State access functions

Although we have already defined some state access functions for the parallel machine, we will find it convenient to define a few more. Each processor will make one state transition, during which it behaves as if it were a sequential machine. If we make the state gmState a pair consisting of the global component of the current machine state and a single processor state, then we have a superset of the state components of any of the sequential G-machines.

```
> gmState == (pgmGlobalState, pgmLocalState)
```

It follows that we can simply redefine the state access functions we used in the sequential machine to work with the new type of state. Here are the type signatures of the global put functions:

```
> putOutput :: gmOutput -> gmState -> gmState
> putHeap :: gmHeap   -> gmState -> gmState
> putSparks :: gmSparks -> gmState -> gmState
> putStats :: gmStats  -> gmState -> gmState
```

The corresponding get functions have type-signatures:

```
> getOutput :: gmState  -> gmOutput
> getHeap :: gmState  -> gmHeap
> getGlobals :: gmState  -> gmGlobals
> getSparks :: gmState  -> gmSparks
> getStats :: gmState  -> gmStats
```

For access to the components local to a processor we need `put` functions with the following type-signatures:

```
> putCode :: gmCode   -> gmState -> gmState
> putStack :: gmStack  -> gmState -> gmState
> putDump :: gmDump   -> gmState -> gmState
> putVStack :: gmVStack -> gmState -> gmState
> putClock :: gmClock  -> gmState -> gmState
```

The `get` functions have types:

```
> getCode :: gmState  -> gmCode
> getStack :: gmState  -> gmStack
> getDump :: gmState  -> gmDump
> getVStack :: gmState  -> gmVStack
> getClock :: gmState  -> gmClock
```

Exercise 5.1. Write the access functions with the types given above.

5.2.2 The evaluator

The structure of the evaluator `eval` will be familiar; it is the similar to the one used in the G-machine.

```
> eval :: pgmState -> [pgmState]
> eval state = state: restStates
>              where
>              restStates = [],                           gmFinal state
>                         = eval (doAdmin (steps state)), otherwise
```

The difference is that we call `steps` instead of `step`. The `steps` function must run down the list of processors doing a single step on each. The precise sequence of events is:

1. First we extract the addresses that were sparked in the previous call to `steps`, from the state.

2. Next, we turn them into processes. This is labelled `newtasks`.

3. The spark pool component of the state is set to empty.

4. We increment the clock for each processor that is about to execute.

5. Finally, we use `mapAccuml` to perform a sequence of `step` transitions, one for each processor.

```
> steps :: pgmState -> pgmState
> steps state
> = mapAccuml step global' local'
>   where ((out, heap, globals, sparks, stats), local) = state
>         newtasks = [makeTask a | a <- sparks]
>         global'  = (out, heap, globals, [], stats)
>         local'   = map tick (local ++ newtasks)
```

To create a task to evaluate a node at address `addr` you must define a `makeTask` function. For machines based on G-machines 2 or 3 this will be:

```
> makeTask :: addr -> pgmLocalState
> makeTask addr = ([Unwind], [addr], [], [], 0)
```

For later marks of the G-machine we use:

```
> makeTask addr = ([Eval], [addr], [], [], 0)
```

Incrementing the clock component of a processor is accomplished using `tick`.

```
> tick (i, stack, dump, vstack, clock) = (i, stack, dump, vstack, clock+1)
```

The machine has terminated when there are no more sparks in the spark pool, and there are no more processors executing tasks.

```
> gmFinal :: pgmState -> bool
> gmFinal s = second s = [] & pgmGetSparks s = []
```

We use the `step` function to perform a single step on a processor.

```
> step :: pgmGlobalState -> pgmLocalState -> gmState
> step global local = dispatch i (putCode is state)
>                     where (i:is) = getCode state
>                           state = (global, local)
```

The `doAdmin` function eliminates processors that have finished executing. A processor has finished when the code component is empty. When this is the case, we must update the statistics component of the state, with the number of instructions it took the processor to complete its task.

```
> doAdmin :: pgmState -> pgmState
> doAdmin ((out, heap, globals, sparks, stats), local)
> = ((out, heap, globals, sparks, stats'), local')
>   where (local', stats') = foldr filter ([], stats) local
>         filter (i, stack, dump, vstack, clock) (local, stats)
>         = (local, clock:stats),                       i = []
>         = ((i, stack, dump, vstack, clock): local, stats), otherwise
```

We now consider the new instruction transitions.

The transition for the Par instruction

The only new instruction that must be added is `Par`. Its effect is to mark the node at the top of the stack so that the machine may create a task to evaluate the node to WHNF. To do this, the instruction must modify the global component of the state by adding the address of the node to the spark pool.

$$(5.1)\qquad \boxed{\begin{array}{rllll} & \langle \mathbf{h} & \mathbf{m} & \mathbf{t} \rangle & \langle \mathtt{Par} : \mathbf{i} \quad \mathbf{a} : \mathbf{s} \rangle \\ \Longrightarrow & \langle \mathbf{h} & \mathbf{m} & \mathbf{a} : \mathbf{t} \rangle & \langle \qquad \mathbf{i} \qquad \mathbf{s} \rangle \end{array}}$$

The first tuple – consisting of **h**, **m** and **t** – is the global state component, with **h**, **m** and **t** being the heap, global address map and spark pool respectively. The second tuple – consisting of an instruction stream and a stack – is a particular task's local state; depending on the version of the G-machine you have used as a basis you may need to add other components to the local state.

The effect of `Par` is to add the address **a** to the spark pool. It is implemented as follows:

```
> par :: gmState -> gmState
```

Exercise 5.2. Modify `showInstruction`, `dispatch` and `instruction` so that `Par` is correctly handled.

5.2.3 Compiling a program

A simple compiler can be constructed for a parallel machine based on any of the sequential machines (except the Mark 7), by providing a compiled primitive for the `par` function. More extensive modifications are required for the Mark 7 based machines.

The other modification required lies in the `compile` function, where the various components are now to be found in different locations, and of course there are now a number of processors. The new definition is:

```
> compile :: coreProgram -> pgmState
> compile program
> = (([], heap, globals, [], []), [initialTask addr])
>   where (heap, globals) = buildInitialHeap program
>         addr            = aLookup globals "main" (error "main undefined")
```

This sets the global component to hold the heap and global map, as used in the sequential G-machine. We also place a task in the local component, to initiate the execution.

```
> initialTask :: addr -> pgmLocalState
> initialTask addr = (initialCode, [addr], [], [], 0)
```

If you use the Mark 2 or Mark 3 sequential G-machine as a basis you need to define `initialCode` as:

```
> initialCode :: gmCode
> initialCode = [Unwind]
```

For the Mark 4 or Mar 5 machine this is changed to:

```
> initialCode = [Eval]
```

And, to deal with data structures, the Mark 6 and Mark 7 machine has the following `initialCode`:

```
> initialCode = [Eval, Print]
```

We now consider how to add `par` to the primitive functions of the machine. We begin by considering those machines based on sequential G-machines Marks 2 through to 6.

Using the Marks 2–6 G-machine as a basis

We need to include the following in the definition of `compiledPrimitives`:

```
> ("par", 2, [Push 1, Push 1, Mkap, Push 2, Par, Update 2, Pop 2, Unwind])
```

This rather cryptic piece of code performs the following task when the function `par` is applied to the two arguments: $\mathbf{E}_1$ and $\mathbf{E}_2$.

1. First we construct the application of $\mathbf{E}_1$ to $\mathbf{E}_2$; this is the job of the sequence:

   ```
   [Push 1, Push 1, Mkap]
   ```

2. Next, `Push 2` makes a copy of the pointer to $\mathbf{E}_2$. The `Par` instruction then adds this address to the spark pool.
3. Finally, we perform the usual updating and tidying-up after an instantiation.

Using the Mark 7 G-machine as a basis

As mentioned above, this is a slightly trickier operation. We need to modify the compiler functions `compileR` and `compileE` to recognise the special cases involving `par`. First `compileR` needs the following case added:

```
> compileR (EAp (EAp (EVar "par") e1) e2) args
> = compileC e2 args ++ [Push 0, Par] ++
>   compileC e1 (argOffset 1 args) ++ [Mkap, Update n, Pop n, Unwind]
>   where n = #args
```

This uses the $\mathcal{C}$ scheme to compile `e2`, which is then sparked. The expression `e1` is compiled using the $\mathcal{C}$ scheme, before we make the application node. Finally, we perform the updating and tidying-up of the stack.

Next, we modify `compileE` so that it has a case:

```
> compileE (EAp (EAp (EVar "par") e1) e2) args
> = compileC e2 args ++ [Push 0, Par] ++
>   compileC e1 (argOffset 1 args) ++ [Mkap, Eval]
```

This only differs from the case given for `compileR` because it uses `Eval` to force the application node that is created to WHNF.

With these two modifications to the compiler, it suffices to add the following to the `primitives`:

```
>  ("par", ["x","y"], (EAp (EAp (EVar "par") (EVar "x")) (EVar "y")))
```

Notice that we could use this approach with the Mark 5 or Mark 6 based machines.

Exercise 5.3. Make the modifications to your compiler, so that there is a `par` function defined.

Exercise 5.4. Why do we perform the sparking of the graph for the second argument before constructing the first argument in the special cases for `compileR` and `compileE`?

5.2.4 Printing the results

Once we have computed the states of the machine we control the display of them by using `showResults`. This prints out: the code for the supercombinators, the state transitions and the statistics. It has type:

```
> showResults :: [pgmState] -> [char]
```

To print the supercombinator code we use the same `showSC` functions as that for the G-machine; it has type:

```
> showSC :: pgmState -> (name, addr) -> iseq
```

The function `showState` is used to display the state of local processes during the transitions. Because we have a parallel machine there is now likely to be more than one task executing at once. It has type:

```
> showState :: pgmState -> iseq
```

Two other functions need to be modified: `showStats` and `showOutput`. They have types:

```
> showStats :: pgmState -> iseq
> showOutput :: gmOutput -> iseq
```

Exercise 5.5. Modify the functions: `showResults`, `showSC`, `showState`, `showStats` and `showOutput`.

Define a new display function `showSparks` with type:

```
> showSparks :: gmSparks -> iseq
```

Exercise 5.6. Try running the parallel G-machine on the following program.

```
main = par (S K K) (S K K 3)
```

How long does it take in machine cycles? How long does it take for the equivalent sequential program:

```
main = S K K (S K K 3)
```

Exercise 5.7. What happens when we run the program:

```
main = par I (I 3)
```

Is the use of `par` justified in this program?

5.3 Mark 2: The evaluate-and-die model

A problem with the Mark 1 machine is that it risks creating many tasks to reduce the same node in the heap, thereby duplicating the work done by the machine. The way to prevent this is to *lock* the nodes during unwinding. This will *block* any other task that encounters the same node. We must also remember to *unlock* nodes once they become free again; this allows blocked tasks to resume.

The only instruction that causes an unlocked node to become locked is `Unwind`; it does this to each node on the spine that it encounters. The reason we choose this instruction, rather than `Eval`, is that it is possible to encounter a locked node part way through unlocking a spine; using `Eval` we would not catch this case. Similarly, the only instruction that will block a task is `Unwind`. After all, it is the only instruction that needs to inspect a node's value.

Conversely, the only instruction that will unlock a locked node is `Update`. A previously locked node is unlocked when it is known to be in WHNF. But we know that *all* nodes in the spine below the redex are in WHNF, when we are about to update the redex, and hence we should unlock all nodes below the root of the redex.

5.3.1 The node data structure

The improvements we intend to incorporate require very few changes to be made to the machine's data structures. First, we must add two sorts of new nodes to the node data type: these will be `NLAp`, the locked application nodes; and `NLGlobal`, locked supercombinator nodes. We will see how they are used in the section on the new instruction transitions.

```
> node ::= NNum num                || Numbers
>        | NAp  addr addr          || Applications
>        | NGlobal num gmCode      || Globals
```

```
>           | NInd addr            || Indirections
>           | NConstr num [addr]   || Constructors
>           | NLAp addr addr       || Locked applications
>           | NLGlobal num gmCode  || Locked globals
```

Exercise 5.8. Rewrite the `showNode` function to deal with the new locked nodes.

5.3.2 The instruction set

The only change that needs to be made is to lock and unlock nodes at the right places. We must lock application nodes and supercombinators with zero arguments as we unwind them. When a node is updated all of the nodes in the spine below it must be unlocked. We use two functions to perform locking and unlocking of heap nodes. The `lock` function turns an unlocked, but possibly updatable, node into a locked one. The nodes that need to be locked are application nodes, and global nodes with no arguments.

```
> lock :: addr -> gmState -> gmState
> lock addr state
> = putHeap (newHeap (hLookup heap addr)) state
>   where
>   heap = getHeap state
>   newHeap (NAp a1 a2)   = hUpdate heap addr (NLAp a1 a2)
>   newHeap (NGlobal n c) = hUpdate heap addr (NLGlobal n c), n = 0
>                         = heap,                             otherwise
```

When we `unlock` a locked application node we need to ensure that the spine that it points to is also unlocked; `unlock` is therefore recursive.

```
> unlock :: addr -> gmState -> gmState
> unlock addr state
> = newState (hLookup heap addr)
>   where
>   heap = getHeap state
>   newState (NLAp a1 a2)
>     = unlock a1 (putHeap (hUpdate heap addr (NAp a1 a2)) state)
>   newState (NLGlobal n c)
>     = putHeap (hUpdate heap addr (NGlobal n c)) state
>   newState n = state
```

The new step transitions for `Unwind` and `Update` should be defined in terms of `lock` and `unlock`; these transitions are now defined. In the Rule 5.2, we see that apart from locking the application node – which is represented by `*NAp` – `Unwind` has the same transition as it did in the Mark 1 machine.

$$(5.2)\quad \boxed{\begin{array}{lllll} & \langle \mathbf{h}[\mathbf{a} : \texttt{NAp}\ \mathbf{a}_1\ \mathbf{a}_2] & \mathbf{m} & \mathbf{t}\rangle & \langle[\texttt{Unwind}] \quad \mathbf{a} : \mathbf{s}\rangle \\ \Longrightarrow & \langle \mathbf{h}[\mathbf{a} : \texttt{*NAp}\ \mathbf{a}_1\ \mathbf{a}_2] & \mathbf{m} & \mathbf{t}\rangle & \langle[\texttt{Unwind}] \quad \mathbf{a}_1 : \mathbf{a} : \mathbf{s}\rangle \end{array}}$$

The same is also true of the transition rule for Unwind when it has a supercombinator of arity zero on top of the stack. In this case *NGlobal is a locked global node.

$$\text{(5.3)} \quad \boxed{\begin{array}{rlll} & \langle \mathbf{h}[\mathbf{a} : \texttt{NGlobal}\ 0\ \mathbf{c}] & \mathbf{m} \quad \mathbf{t}\rangle & \langle[\texttt{Unwind}] \quad \mathbf{a} : \mathbf{s}\rangle \\ \Longrightarrow & \langle \mathbf{h}[\mathbf{a} : \texttt{*NGlobal}\ 0\ \mathbf{c}] & \mathbf{m} \quad \mathbf{t}\rangle & \langle \qquad\quad \mathbf{c} \quad \mathbf{a} : \mathbf{s}\rangle \end{array}}$$

In the new transition rule for Update, when we update the root of the redex, whose address is **a**, we must unlock all of the nodes in the spine descending from **a**. The transition rule is therefore:

$$\text{(5.4)} \quad \boxed{\begin{array}{rlll} & \langle \mathbf{h} \left[\begin{array}{lcl} \mathbf{a}'_1 & : & \texttt{NGlobal}\ \mathbf{n}\ \mathbf{c} \\ \mathbf{a}'_2 & : & \texttt{*NAp}\ \mathbf{a}'_1\ \mathbf{a}_1 \\ & & \quad\cdots \\ \mathbf{a}'_{\mathbf{n}-1} & : & \texttt{*NAp}\ \mathbf{a}'_{\mathbf{n}-2}\ \mathbf{a}_{\mathbf{n}-2} \\ \mathbf{a}_{\mathbf{n}} & : & \texttt{*NAp}\ \mathbf{a}'_{\mathbf{n}-1}\ \mathbf{a}_{\mathbf{n}-1} \end{array}\right] & \mathbf{m} \quad \mathbf{t}\rangle & \langle \texttt{Update}\ \mathbf{n} : \mathbf{i} \quad \mathbf{a} : \mathbf{a}_1 : \ldots : \mathbf{a}_{\mathbf{n}} : \mathbf{s}\rangle \\ \Longrightarrow & \langle \mathbf{h} \left[\begin{array}{lcl} \mathbf{a}'_1 & : & \texttt{NGlobal}\ \mathbf{n}\ \mathbf{c} \\ \mathbf{a}'_2 & : & \texttt{NAp}\ \mathbf{a}'_1\ \mathbf{a}_1 \\ & & \quad\cdots \\ \mathbf{a}'_{\mathbf{n}-1} & : & \texttt{NAp}\ \mathbf{a}'_{\mathbf{n}-2}\ \mathbf{a}_{\mathbf{n}-2} \\ \mathbf{a}_{\mathbf{n}} & : & \texttt{NInd}\ \mathbf{a} \end{array}\right] & \mathbf{m} \quad \mathbf{t}\rangle & \langle \qquad\qquad \mathbf{i} \qquad \mathbf{a}_1 : \ldots : \mathbf{a}_{\mathbf{n}} : \mathbf{s}\rangle \end{array}}$$

Exercise 5.9. Modify the definitions of the transition functions unwind and update. You should use the lock and unlock functions in your code.

You will also need to 'look through' locked nodes on rearranging the stack, so getArg becomes:

```
> getArg (NLAp a1 a2) = a2
```

Exercise 5.10. Try running this program on your machine:

```
main = twice' (twice' (twice' (S K K))) 3
twice' f x = par f (f x)
```

Exercise 5.11. A divide-and-conquer program executes a par instruction in each running process every thirty instructions. The processes do not die. How many simulated clock ticks pass before we have one task for each electron in the universe? (Hint: there are approximately 10^{85} electrons in the universe.)

Exercise 5.12. If a processor costs £0.01, how long can the program of Exercise 5.11 run before nobody can afford the machine? (Hint: the US federal budget is approximately £5×10^{12}.)

5.4 Mark 3: A realistic parallel G-machine

As Exercises 5.11 and 5.12 will have shown there are physical and economic limitations to the amount of parallelism available in the real world. The model of parallelism that we are using is not very realistic. We are creating a new processor to execute each parallel task, and in the real world we will very quickly run out of resources.

5.4.1 Scheduling policy

A more realistic model must involve restricting the amount of parallelism to that provided by the hardware. This is easily accomplished by placing an upper limit on the number of processors that can run at any one time. As a consequence, whenever there are no processors available to execute a task, the task will remain unchanged.

When there are more processors than tasks, some processors will be idle. On the other hand, when the reverse is the case, we will be faced with the problem of deciding which task we will execute next. This decision is called a *scheduling policy.*

5.4.2 Conservative and speculative parallelism

Some tasks will be more important than others. We can usefully classify tasks into one of two groups:

- tasks whose results will *definitely* be needed; and
- tasks whose results *may* be needed.

We refer to tasks in the first category as *conservative tasks*, whilst those in the second are termed *speculative tasks.*

If we choose to allow speculative parallel tasks, then we must address issues of *priority* in scheduling tasks. That is: we must rank the tasks in order of importance. We must also allow different tasks with the same priority the same amount of computing time. To see why this is desirable, consider evaluating two branches of a case expression – `e1` and `e2` – in parallel with the evaluation of the discriminant expression `e0`.

```
case e0 of
     <1> -> e1
     <2> -> e2
```

Until `e0` has completed its evaluation, we do not know which of `e1` or `e2` will be required; it therefore makes sense to try to evaluate an equal amount of each. This sort of scheduling policy is termed *fair scheduling.* Notice, in this example, that once `e0` *has* completed, one of the tasks (`e1` or `e2`) will become needed, and the other should be killed. The priorities of a task therefore need to be adjusted during the execution of a task.

In this book we make no attempt to implement a machine suitable for speculative parallelism. We make the excuse that this is a 'hard problem', and leave the matter alone. Henceforth, all uses of the `par` primitive are assumed to occur in situations which give rise only to conservative parallelism.

In the Mark 3 parallel G-machine, we will only have a limited number of active tasks within the machine. These are executed by the *processors* in the machine. There will be only a fixed number of processors. This fixed number is: `machineSize`, which we have currently set to `4`.

```
> machineSize :: num
> machineSize = 4
```

The major change to the evaluator lies in the steps function, which does not just add all of the tasks that were created into the machine. Instead it now uses scheduler to pick replacement tasks for any task that cannot proceed.

1. First, we extract the sparks in the task pool from the global state component.
2. New tasks are then created for the sparks, and are added to the already executing tasks.
3. The scheduler function then selects which tasks to execute.

Here is the way we code steps:

```
> steps :: pgmState -> pgmState
> steps state
> = scheduler global' local'
>   where ((out, heap, globals, sparks, stats), local) = state
>         newtasks = [makeTask a | a <- sparks]
>         global'  = (out, heap, globals, [], stats)
>         local'   = local ++ newtasks
```

The scheduling policy is very simple: we select the first machineSize tasks and run them. These tasks are then placed at the end of the scheduling queue. This scheduling policy is usually called a *round-robin* scheduling policy.

```
> scheduler :: pgmGlobalState -> [pgmLocalState] -> pgmState
> scheduler global tasks
> = (global', nonRunning ++ tasks')
>   where running    = map tick (take machineSize tasks)
>         nonRunning = drop machineSize tasks
>         (global', tasks') = mapAccuml step global running
```

Exercise 5.13. What happens if the tasks that are executed are not placed at the end of the scheduling queue for the next step. (Hint: try it!)

Exercise 5.14. One improvement we can make is to create tasks from the spark pool only when there are idle processors. Modify steps to do this.

Exercise 5.15. Another improvement is only to schedule tasks that can proceed. At the moment we schedule tasks to evaluate nodes that are already in WHNF. We also schedule tasks that are blocked because they are attempting to unwind locked nodes. Modify scheduler so that this no longer happens.

Exercise 5.16. Investigate the use of other scheduling strategies. For example, try scheduling the last task in the task pool.

Does the scheduling strategy make any difference to the execution time of the program?

5.5 Mark 4: A better way to handle blocking

So far we have left blocked tasks in the machine's local state, and required the `scheduler` function to select runnable tasks. This means that the `scheduler` function may have to skip over a considerable number of blocked tasks before coming across one that it can run.

We can do better than this! It would be a much better idea to attach a blocked task to the node that caused it to block. Because a locked node can cause an arbitrary number of tasks to block, we will need to allow a list of tasks to be placed on the locked node. We call this the *pending list*.

How do we use the pending list?

1. When a node is locked, it has its pending list set to `[]`.
2. When a task encounters a locked node, the task places itself on the pending list of the locked node.
3. When a locked node is unlocked, all of the tasks in its pending list become runnable, and are transferred to the machine's local state.

To implement the Mark 4 machine we must make the following changes to the data structures of the machine.

5.5.1 Data structures

First, each locked node must now have a pending list; this means that the `node` data type is now:

```
> node ::= NNum num                              || Numbers
>        | NAp  addr addr                        || Applications
>        | NGlobal num gmCode                    || Globals
>        | NInd addr                             || Indirections
>        | NConstr num [addr]                    || Constructors
>        | NLAp addr addr pgmPendingList         || Locked applications
>        | NLGlobal num gmCode pgmPendingList    || Locked globals
```

A pending list is just a list of tasks. The tasks in a locked node's pending list will be those that have been blocked on the locked node.

```
> pgmPendingList == [pgmLocalState]
```

Exercise 5.17. Modify `showNode` to work with the new definition of `node`.

The other change is to the type of `gmSparks`. Instead of being a list of addresses – as it was in previous parallel G-machines – this is now a list of tasks.

```
> gmSparks == [pgmLocalState]
```

Exercise 5.18. Modify the `par` transition so that it places tasks into the spark pool and not addresses. You should modify `showSparks` function to print the number of tasks in the spark pool, and the `steps` function will need to be modified because it no longer needs to turn items in the spark pool into tasks.

5.5.2 Locking and unlocking

We have been building up to a new way to handle locking and unlocking of nodes. Let us first consider what happens when a locked node is about to be updated. The `Update` instruction will be implemented using a call to `unlock`. Each node in the spine of the expression about to be overwritten will need to have the tasks in its pending list transferred to the spark pool. To do this we define the following function that transfers tasks to the spark pool:

```
> emptyPendingList :: [pgmLocalState] -> gmState -> gmState
> emptyPendingList tasks state
> = putSparks (tasks ++ getSparks state) state
```

Exercise 5.19. Modify the `unlock` function so that it empties the tasks in the pending lists into the spark pool.

The locking operation occurs as part of the `Unwind` instruction. As previously, we use the `lock` function to perform the locking operation. Now it must give each newly locked node an empty pending list.

Exercise 5.20. Modify the `lock` function so that it gives locked nodes an empty pending list.

Finally, we discuss what happens when a task attempts to unwind a locked node. Clearly, we place the task onto the node's pending list. But what do we replace the task with? Remember that the type of `step` is:

```
> step :: gmState -> gmState
```

The solution we have adopted is to replace the task with an `emptyTask`:

```
> emptyTask :: pgmLocalState
> emptyTask = ([], [], [], [], 0)
```

So we need two new transitions for `Unwind`. We begin with the one for locked application nodes, in which we see that the task is placed on the node's pending list and we see that the task is replaced by the `emptyTask`.

$$(5.5)\quad \boxed{\begin{array}{lllll} & \langle \mathbf{h}[\mathbf{a} : {*}\mathtt{NAp}\ \mathbf{a}_1\ \mathbf{a}_2\ \mathbf{pl}] & \mathbf{m} & \mathbf{t}\rangle & \langle[\mathtt{Unwind}]\quad \mathbf{a} : \mathbf{s}\rangle \\ \Longrightarrow & \langle \mathbf{h}[\mathbf{a} : {*}\mathtt{NAp}\ \mathbf{a}_1\ \mathbf{a}_2\ \langle[\mathtt{Unwind}], \mathbf{a} : \mathbf{s}\rangle : \mathbf{pl}] & \mathbf{m} & \mathbf{t}\rangle & \mathtt{emptyTask} \end{array}}$$

The rule for locked global nodes is similar: we see that the task is placed onto the node's pending list, and is itself replaced by the `emptyTask`.

$$(5.6)\quad \boxed{\begin{array}{lllll} & \langle \mathbf{h}[\mathbf{a} : *\texttt{NGlobal}\ 0\ \mathbf{c}\ \mathbf{pl}] & \mathbf{m} & \mathbf{t}\rangle & \langle[\texttt{Unwind}]\quad \mathbf{a} : \mathbf{s}\rangle \\ \Longrightarrow & \langle \mathbf{h}[\mathbf{a} : *\texttt{NGlobal}\ 0\ \mathbf{c}\ \langle[\texttt{Unwind}], \mathbf{a} : \mathbf{s}\rangle : \mathbf{pl}] & \mathbf{m} & \mathbf{t}\rangle & \texttt{emptyTask} \end{array}}$$

Exercise 5.21. Modify the `unwind` function to implement the new transitions for the `Unwind` instruction. You will also need to make the `getArg` function:

```
> getArg (NLAp a1 a2 pl) = a2
```

Exercise 5.22. Modify the `scheduler` function to place non-running tasks into the spark pool.

Exercise 5.23. Modify the `doAdmin` function to filter out `emptyTask`'s from the local state.

5.6 Conclusions

This chapter has shown that, in principle, a shared memory implementation of lazy functional languages is straightforward. Of course, we have also seen that there are payoffs to be had by carefully considering optimisations to the simple scheme we used initially in the Mark 1 machine. In all of our parallel machines, the graph acts as a communication and synchronisation medium; and in the Mark 2 and Mark 3 machines, individual processes will be blocked when trying to access locked nodes in the heap.

So where are the current challenges in the parallel implementation of lazy functional languages? The mechanisms for parallelism included in this book do not handle the deletion of processes. If speculative parallelism is going to be used then realistic implementations will need to deal with this problem. On the other hand, finding the non-speculative parallelism is often difficult, and in large programs this may even be intractable. Attempts have been made to use abstract interpretation for this purpose, and although the results look promising, they should be regarded tentatively.

One final area that we have not covered is that of distributed memory parallel machines. Again, in principle they are similar to shared memory machines, but the practicalities are quite different. Arranging the message passing so as to avoid deadlock is something of a black art.

Chapter 6

Lambda Lifting

6.1 Introduction

In this chapter[1] we will be looking at ways to extend the set of programs that are acceptable to the machines we have looked at previously in the book. The extension that we introduce is to allow *local function definitions*. We are then faced with alternative approaches:

- add a mechanism to the machines to deal with environments; or
- transform the program so that there are no local function definitions; instead all functions are defined as supercombinators.

In this book we have always assumed that the second approach would be taken.

This chapter is also an appropriate point at which to introduce the concept of *full laziness*. Again, this desirable optimisation of functional languages is achieved using a program transformation.

6.2 Improving the `expr` data type

Before we begin the program proper, we must import the language and utilities modules.

```
> %include "language"
> %include "utils"
```

Unfortunately, the data types defined there (`coreExpr`, `coreProgram` and so on) are insufficiently flexible for our needs in this chapter, so we will attend to this problem first. Many compiler passes add information to the abstract syntax tree, and we need a systematic way to represent this information. Examples of analyses which generate this sort of information are: free-variable analysis, binding level analysis, type inference, strictness analysis and sharing analysis.

The most obvious way to add such information is to add a new constructor for annotations to the `expr` data type, thus:

[1]Some of the material in this chapter was first published in [Peyton Jones and Lester 1991].

```
> expr * = EVar name
>          | ...
>          | EAnnot annotation (expr *)
```

together with an auxiliary data type for annotations, which can be extended as required:

```
> annotation ::= FreeVars (set name)
>              | Level num
```

This allows annotations to appear freely throughout the syntax tree, which appears admirably flexible. In practice, it suffers from two major disadvantages:

- It is easy enough to *add* annotation information in the form just described, but writing a compiler pass which *uses* information placed there by a previous pass is downright awkward. Suppose, for example, that a pass wishes to use the free-variable information left at every node of the tree by a previous pass. Presumably this information is attached immediately above every node, but the data type would permit several annotation nodes to appear above each node, and worse still none (or more than one) might have a free-variable annotation.

  Even if the programmer is prepared to certify that there is exactly one annotation node above every tree node, and that it is a free-variable annotation, the implementation will still perform pattern matching to check these assertions when extracting the annotation.

  Both of these problems, namely the requirement for uncheckable programmer assertions and some implementation overhead, are directly attributable to the fact that every annotated tree has the rather uninformative type `expr`, which says nothing about which annotations are present.

- The second major problem is that further experimentation reveals that *two* distinct forms of annotation are required. The first annotates expressions as above, but the second annotates the binding occurrences of variables; that is, the occurrences on the left-hand sides of `let(rec)` definitions, and the bound variables in lambda abstractions or case expressions. We will call these occurrences *binders*. An example of the need to annotate binders comes in type inference, where the compiler infers a type for each binder, as well as for each sub-expression.

  It is possible to use the expression annotation to annotate binders, but it is clumsy and inconvenient to do so.

We will address the second problem first, since it has an easy solution. Recall from Section 1.3 that the `expr` type was *parameterised* with respect to the type of its binders; we repeat its definition here as a reminder:

```
> expr *
>  ::=  EVar name                        || Variables
>       | ENum num                       || Numbers
>       | EConstr num num                || Constructor tag arity
>       | EAp (expr *) (expr *)          || Applications
>       | ELet                           || Let(rec) expressions
```

```
>           isRec                       ||   boolean with True = recursive,
>           [(*, expr *)]               ||   Definitions
>           (expr *)                    ||   Body of let(rec)
>       | ECase                         || Case expression
>           (expr *)                    ||   Expression to scrutinise
>           [alter *]                   ||   Alternatives
>       | ELam [*] (expr *)             || Lambda abstractions
```

The type `coreExpr` is a specialised form of `expr` in which the binders are of type `name`. This is expressed using a type synonym (also repeated from Section 1.3):

```
> coreExpr == expr name
```

The advantage of parameterising `expr` is that we can also define other specialised forms. For example, `typedExpr` is a data type in which binders are names annotated with a type:

```
> typedExpr = expr (name, typeExpr)
```

where `typeExpr` is a data type representing type expressions.

Returning to annotations on expressions, we can reuse the same technique by parameterising the data type of expressions with respect to the annotation type. We want to have an annotation on every node of the tree, so one possibility would be to add an extra field to every constructor with the annotation information. This is inconvenient if, for example, you simply want to extract the free-variable information at the top of a given expression without performing case analysis on the root node. This leads to the following idea:

> *each level of the tree is a pair, whose first component is the annotation, and whose second component is the abstract syntax tree node.*

Here are the corresponding Miranda data type definitions:

```
> annExpr * ** == (**, annExpr' * **)


> annExpr' * ** ::= AVar name
>                   | ANum num
>                   | AConstr num num
>                   | AAp (annExpr * **) (annExpr * **)
>                   | ALet bool [annDefn * **] (annExpr * **)
>                   | ACase (annExpr * **) [annAlt * **]
>                   | ALam [*] (annExpr * **)


> annDefn * ** == (*, annExpr * **)


> annAlt * **  == (num, [*], (annExpr * **))
```

```
> annProgram * ** == [(name, [*], annExpr * **)]
```

Notice the way that the mutual recursion between `annExpr` and `annExpr'` ensures that every node in the tree carries an annotation. The sort of annotations carried by an expression are now manifested in the type of the expression. For example, an expression annotated with free variables has type `annExpr name (set name)`.

It is a real annoyance that `annExpr'` and `expr` have to define two essentially identical sets of constructors. There appears to be no way around this within the Hindley-Milner type system. It would be possible to abandon the `expr` type altogether, because the `expr *` is nearly isomorphic to `annExpr * **`, but there are two reasons why we choose not to do this. Firstly, the two types are not quite isomorphic, because the latter distinguishes `((), ⊥ )` from ⊥ while the former does not. Secondly (and more seriously), it is very tiresome to write all the `()`'s when building and pattern matching on trees of type `annExpr * **`.

This completes our development of the central data type. The discussion has revealed some of the strengths, and a weakness, of the algebraic data types provided by modern functional programming languages.

Exercise 6.1. Our present pretty-printing function, `pprint`, defined in Section 1.5, is only able to print `corePrograms`. In order to print out intermediate stages in the lambda lifter we will need a function `pprintGen` which can display values of type `program *`. (The 'Gen' is short for 'generic'.) `pprintGen` needs an extra argument to tell it how to print the binders:

```
> pprintGen :: (* -> iseq)      || function from binders to iseq
>              -> program *     || the program to be formatted
>              -> [char]        || result string
```

For example, once we have written `pprintGen` we can define `pprint` in terms of it:

```
> pprint prog = pprintGen iStr prog
```

Write a definition for `pprintGen`, and its associated functions `pprExprGen`, and so on.

Exercise 6.2. Do a similar job for printing values of type `annProgram * **`. Here you will need two extra arguments, one for formatting the binders and one for formatting the annotations:

```
> pprintAnn :: (* -> iseq)              || function from binders to iseq
>              -> (** -> iseq)          || function from annotations to iseq
>              -> annProgram * **       || program to be displayed
>              -> [char]                || result string
```

Write a definition for `pprintAnn` and its associated auxiliary functions.

6.3 Mark 1: A simple lambda lifter

Any implementation of a lexically scoped programming language has to cope with the fact that a function or procedure may have *free variables*. Unless these are removed in some way, an environment-based implementation has to manipulate linked environment frames, and a reduction-based system is made significantly more complex by the need to perform renaming during substitution. A popular way of avoiding these problems, especially in graph reduction implementations, is to eliminate all free variables from function definitions by means of a transformation known as *lambda lifting*. Lambda lifting is a term coined by [Johnsson 1985], but the transformation was independently developed by [Hughes 1983].

In our context, lambda lifting transforms a Core-language program into an equivalent one in which there are no embedded lambda abstractions. To take a simple example, consider the program

```
f x = let g = \y. x*x + y in (g 3 + g 4)
main = f 6
```

The `\y` abstraction can be removed by defining a new supercombinator `$g` which takes `x` as an extra parameter, but whose body is the offending abstraction, and replacing the `\y` abstraction with an application of `$g`, giving the following set of supercombinator definitions:

```
$g x y = x*x + y
f x = let g = $g x in (g 3 + g 4)
main = f 6
```

How did we decide to make just `x` into the extra parameter to `$g`? We did it because `x` is a *free variable* of the abstraction `\y. x*x + y`:

> ***Definition.*** *An occurrence of a variable* **v** *in an expression* **e** *is said to be* free *in* **e** *if the occurrence is not bound by an enclosing lambda or* `let(rec)` *expression in* **e**.

On the other hand, `y` is not free in `(\y. x*x + y)`, because its occurrence is in the scope of an enclosing lambda abstraction which binds it.

Matters are no more complicated when recursion is involved. Suppose that `g` was recursive, thus:

```
f x = letrec g = \y. cons (x*y) (g y) in g 3
main = f 6
```

Now `x` and `g` are both free in the `\y` abstraction, so the lambda lifter will make them both into extra parameters of `$g`, producing the following set of supercombinators:

```
$g g x y = cons (x*y) (g y)
f x = letrec g = $g g x in g 3
main = f 6
```

Notice that the definition of `g` is still recursive, but the lambda lifter has eliminated the local lambda abstraction. The program is now directly implementable by most compiler back ends; and in particular by all of the abstract machines in this book.

There one final gloss to add: there is no need to treat other top-level functions as extra parameters. For example, consider the program

```
h p q = p*q
f x = let g = \y. (h x x) + y in (g 3 + g 4)
main = f 6
```

Here we do not want to treat `h` as a free variable of the `\y` abstraction, because it is a constant which can be referred to directly from the body of the new `$g` supercombinator. The same applies, of course, to the `+` and `*` functions! In short, only supercombinator arguments, and variables bound by lambda abstractions or `let(rec)` expressions, are candidates for being free variables.

It is worth noting in passing that the lexical-scoping issue is not restricted to functional languages. For example, Pascal allows a function to be declared locally within another function, and the inner function may have free variables bound by the outer scope. On the other hand, the C language does not permit such local definitions. In the absence of side effects, it is simple to make a local function definition into a global one: all we need do is add the free variables as extra parameters, and add these parameters to every call. This is exactly what lambda lifting does.

6.3.1 Implementing a simple lambda lifter

We are now ready to develop a simple lambda lifter. It will take a `coreProgram` and return an equivalent `coreProgram` in which there are no occurrences of the `ELam` constructor.

```
> lambdaLift :: coreProgram -> coreProgram
```

The lambda lifter works in three passes:

- First, we annotate every node in the expression with its free variables. This is used by the following pass to decide which extra parameters to add to a lambda abstraction. The `freeVars` function has type

  ```
  > freeVars :: coreProgram -> annProgram name (set name)
  ```

  The type `set *` is a standard abstract data type for sets, whose definition is given in Appendix A.4.

- Second, the function `abstract` abstracts from each lambda abstraction `\`$\mathbf{x}_1 \ldots \mathbf{x_n}$`.`$\mathbf{e}$ its free variables $\mathbf{v}_1 \ldots \mathbf{v_m}$, replacing the lambda abstraction with an expression of the form

  (`let sc = \`$\mathbf{v}_1 \ldots \mathbf{v_m}\ \mathbf{x}_1 \ldots \mathbf{x_n}$`. e in sc`) $\mathbf{v}_1 \ldots \mathbf{v_m}$

  We could use a direct application of the lambda abstraction to the free variables, but we need to give the new supercombinator a name, so we take the first step here by always giving it the name `sc`. For example, the lambda abstraction

  ```
  (\x. y*x + y*z)
  ```

  would be transformed to

```
(let sc = (\y z x.  y*x + y*z) in sc) y z
```

`abstract` has the type signature:

```
> abstract :: annProgram name (set name) -> coreProgram
```

Notice, from the type signature, that `abstract` removes the free variable information, which is no longer required.

- Now we traverse the program giving a unique name to each variable. This will have the effect of making unique all the `sc` variables introduced by the previous pass. Indeed, the sole purpose of introducing the extra `let` expressions in the first place was to give each supercombinator a name which could then be made unique. As a side effect, all other names in the program will be make unique, but this does not matter, and it will turn out to be useful later.

  ```
  > rename :: coreProgram -> coreProgram
  ```

- Finally, `collectSCs` collects all the supercombinator definitions into a single list, and places them at the top level of the program.

  ```
  > collectSCs :: coreProgram -> coreProgram
  ```

The lambda lifter itself is the composition of these three functions:

```
> lambdaLift = collectSCs . rename . abstract . freeVars
```

To make it easier to see what is happening we define the a function `runS` (the 'S' stands for 'simple') to integrate the parser and printer:

```
> runS = pprint . lambdaLift . parse
```

It would of course be possible to do all the work in a single pass, but the modularity provided by separating them has a number of advantages: each individual pass is easier to understand, the passes may be reusable (for example, we reuse `freeVars` below) and modularity makes it easier to change the algorithm somewhat.

As an example of the final point, some compilers are able to generate better code by omitting the `collectSCs` pass, because more is then known about the context in which the supercombinator is applied [Peyton Jones 1991]. For example, consider the following expression, which might be produced by the `abstract` pass:

```
let f = (\v x. v-x) v
in ...f...f...
```

Here abstract has removed v as a free variable from the \x abstraction[2]. Rather than compiling the supercombinator independently of its context, a compiler could construct a closure for f, whose code accesses v directly from the closure and x from the stack. The calls to f thus do not have to move v onto the stack. The more free variables there are the more beneficial this becomes. Nor do the calls to f become less efficient because the definition is a local one; the compiler can see the binding for f and can jump directly to its code.

In the following sections we give definitions for each of these passes. We omit the equations for case expressions, which appear as Exercise 6.4.

6.3.2 Free variables

The core of the free-variable pass is function freeVars_e which has type

```
> freeVars_e :: (set name)                    || Candidates for free variables
>               -> coreExpr                   || Expression to annotate
>               -> annExpr name (set name)    || Annotated result
```

Its first argument is the set of local variables which are in scope; these are the possible free variables. The second argument is the expression to be annotated, and the result is the annotated expression. The main function freeVars just runs down the list of supercombinator definitions, applying freeVars_e to each:

```
> freeVars prog = [ (name, args, freeVars_e (setFromList args) body)
>                 | (name, args, body) <- prog
>                 ]
```

The freeVars_e function runs over the expression recursively; in the case of numbers there are no free variables, so this is what is returned in the annotated expression.

```
> freeVars_e lv (ENum k)      = (setEmpty, ANum k)
```

In the case of a variable, we check to see whether it is in the set of candidates to decide whether to return the empty set or a singleton set:

```
> freeVars_e lv (EVar v)      = (setSingleton v, AVar v),  setElementOf v lv
>                             = (setEmpty, AVar v),        otherwise
```

The case for applications is straightforward: we first annotate the expression e1 with its free variables, then annotate e2, returning the union of the two sets of free variables as the free variables of EAp e1 e2.

```
> freeVars_e lv (EAp e1 e2)
> = (setUnion (freeVarsOf e1') (freeVarsOf e2'), AAp e1' e2')
>   where e1'            = freeVars_e lv e1
>         e2'            = freeVars_e lv e2
```

[2]We are ignoring the let expression which abstract introduces to name the supercombinator.

In the case of a lambda abstractions we need to add the `args` to the local variables passed in, and subtract them from the free variables passed out:

```
> freeVars_e lv (ELam args body)
> = (setSubtraction (freeVarsOf body') (setFromList args), ALam args body')
>   where body'          = freeVars_e new_lv body
>         new_lv         = setUnion lv (setFromList args)
```

The equation for `let(rec)` expressions has rather a lot of plumbing, but is quite straightforward. The local variables in scope that are passed to the body is `body_lv`; the set of local variables passed to each right-hand side is `rhs_lv`. Next we annotate each right-hand side with its free variable set, giving `rhss'`, from this we can construct the annotated definitions: `defns'`. The annotated body of the `let(rec)` is `body'`. The free variables of the definitions is calculated to be `defnsFree`, and those of the body are `bodyFree`.

```
> freeVars_e lv (ELet is_rec defns body)
> = (setUnion defnsFree bodyFree, ALet is_rec defns' body')
>   where binders        = bindersOf defns
>         binderSet      = setFromList binders
>         body_lv        = setUnion lv binderSet
>         rhs_lv         = body_lv,     is_rec
>                        = lv,          otherwise
>
>         rhss'          = map (freeVars_e rhs_lv) (rhssOf defns)
>         defns'         = zip2 binders rhss'
>         freeInValues   = setUnionList (map freeVarsOf rhss')
>         defnsFree      = setSubtraction freeInValues binderSet, is_rec
>                        = freeInValues,                          otherwise
>         body'          = freeVars_e body_lv body
>         bodyFree       = setSubtraction (freeVarsOf body') binderSet
```

The function `zip2` in the definition of `defns'` is a standard function which takes two lists and returns a list consisting of pairs of corresponding elements of the argument lists. The set operations `setUnion`, `setSubtraction` and so on are defined in the utilities module, whose interface is given in Appendix A.4.

We postpone dealing with `case` and constructor expressions:

```
> freeVars_e lv (ECase e alts) = freeVars_case lv e alts
> freeVars_e lv (EConstr t a) = error "freeVars_e: no case for constructors"
> freeVars_case lv e alts = error "freeVars_case: not yet written"
```

`freeVarsOf` and `freeVarsOf_alt` are simple auxiliary functions:

```
> freeVarsOf :: annExpr name (set name) -> set name
> freeVarsOf (free_vars, expr) = free_vars
```

```
> freeVarsOf_alt :: annAlt name (set name) -> set name
> freeVarsOf_alt (tag, args, rhs)
> = setSubtraction (freeVarsOf rhs) (setFromList args)
```

6.3.3 Generating supercombinators

The next pass merely replaces each lambda abstraction, which is now annotated with its free variables, with a new abstraction (the supercombinator) applied to its free variables.

```
> abstract prog = [ (sc_name, args, abstract_e rhs)
>                 | (sc_name, args, rhs) <- prog
>                 ]
```

As usual, we define an auxiliary function `abstract_e` to do most of the work:

```
> abstract_e :: annExpr name (set name) -> coreExpr
```

It takes an expression annotated with the free variable information and returns an expression with each lambda abstraction replaced by a new abstraction applied to the free variables. There is little to say about the first four cases, they just recursively abstract each expression.

```
> abstract_e (free, AVar v)    = EVar v
> abstract_e (free, ANum k)    = ENum k
> abstract_e (free, AAp e1 e2) = EAp (abstract_e e1) (abstract_e e2)
> abstract_e (free, ALet is_rec defns body)
> =  ELet is_rec [ (name, abstract_e body) | (name, body) <- defns]
>                (abstract_e body)
```

The function `foldll` is a standard function, defined in Appendix A.5; given a dyadic function $\oplus$, a value $\mathbf{b}$, and a list $\mathbf{xs} = [\mathbf{x}_1, ..., \mathbf{x}_\mathbf{n}]$, `foldll` $\oplus$ $\mathbf{b}$ $\mathbf{xs}$ computes $(\ldots((\mathbf{b} \oplus \mathbf{x}_1) \oplus \mathbf{x}_2) \oplus \ \ldots \mathbf{x}_\mathbf{n})$. Notice the way that the free-variable information is discarded by the pass, since it is no longer required.

The final case we show is the heart of the `abstract_e` function. First we create a list of free variables: `fvList`. We recall that there is no ordering implicit in a set; the function `setToList` has induced an ordering on the elements, but we do not much care what order this is. Next we make a new supercombinator. This involves

1. applying `abstract_e` to the body of the lambda expression; and
2. augmenting the argument list, by prefixing the original one with the free-variable list.

Next, to allow the `collectSCs` pass to detect this new supercombinator, we wrap it into a `let` expression. Finally, we apply the new supercombinator to each free variable in turn.

```
> abstract_e (free, ALam args body)
> = foldll EAp sc (map EVar fvList)
```

```
>   where
>   fvList = setToList free
>   sc = ELet nonRecursive [("sc",sc_rhs)] (EVar "sc")
>   sc_rhs = ELam (fvList ++ args) (abstract_e body)
```

`case` expressions and constructors are deferred:

```
> abstract_e (free, AConstr t a) = error "abstract_e: no case for Constr"
> abstract_e (free, ACase e alts) = abstract_case free e alts
> abstract_case free e alts = error "abstract_case: not yet written"
```

It is worth observing that `abstract_e` treats the two expressions `(ELam args1 (ELam args2 body))` and `(ELam (args1++args2) body)` differently. In the former case, the two abstractions will be treated separately, generating two supercombinators, while in the latter only one supercombinator is produced. It is clearly advantageous to merge directly nested `ELam`s before performing lambda lifting. This is equivalent to the η-abstraction optimisation noted by [Hughes 1983].

6.3.4 Making all the variables unique

Next, we need to make each variable so that all the `sc` variables introduced by `abstract` are unique. The auxiliary function, `rename_e`, takes an environment mapping old names to new names, a name supply and an expression. It returns a depleted name supply and a new expression.

```
> rename_e :: assoc name name                    || Binds old names to new
>             -> nameSupply                      || Name supply
>             -> coreExpr                        || Input expression
>             -> (nameSupply, coreExpr)          || Depleted supply and result
```

Now we can define `rename` in terms of `rename_e`, by applying the latter to each supercombinator definition, plumbing the name supply along with `mapAccuml`.

```
> rename prog
> = second (mapAccuml rename_sc initialNameSupply prog)
>   where
>   rename_sc ns (sc_name, args, rhs)
>   = (ns2, (sc_name, args', rhs'))
>     where
>     (ns1, args', env) = newNames ns args
>     (ns2, rhs') = rename_e env ns1 rhs
```

The function `newNames` takes a name supply and a list of names as its arguments. It allocates a new name for each old one from the name supply, returning the depleted name supply, a list of new names and an association list mapping old names to new ones.

```
> newNames :: nameSupply -> [name] -> (nameSupply, [name], assoc name name)
> newNames ns old_names
> = (ns', new_names, env)
>   where
>   (ns', new_names) = getNames ns old_names
>   env = zip2 old_names new_names
```

The definition of rename_e is now straightforward, albeit dull. When we meet a variable, we look it up in the environment. For top-level functions and built-in functions (such as +) we will find no substitution for it in the environment, so we just use the existing name:

```
> rename_e env ns (EVar v)      = (ns, EVar (aLookup env v v))
```

Numbers and applications are easy.

```
> rename_e env ns (ENum n)      = (ns, ENum n)
> rename_e env ns (EAp e1 e2)
> = (ns2, EAp e1' e2')
>   where
>   (ns1, e1') = rename_e env ns e1
>   (ns2, e2') = rename_e env ns1 e2
```

When we meet an ELam we need to invent new names for the arguments, using newNames, and augment the environment with the mapping returned by newNames.

```
> rename_e env ns (ELam args body)
> = (ns1, ELam args' body')
>   where
>   (ns1, args', env') = newNames ns args
>   (ns2, body') = rename_e (env' ++ env) ns1 body
```

let(rec) expressions work similarly:

```
> rename_e env ns (ELet is_rec defns body)
> = (ns3, ELet is_rec (zip2 binders' rhss') body')
>   where
>   (ns1, body') = rename_e body_env ns body
>   binders = bindersOf defns
>   (ns2, binders', env') = newNames ns1 binders
>   body_env = env' ++ env
>   (ns3, rhss') = mapAccuml (rename_e rhsEnv) ns2 (rhssOf defns)
>   rhsEnv = body_env,  is_rec
>          = env,       otherwise
```

We leave case expressions as an exercise:

```
> rename_e env ns (EConstr t a) = error "rename_e: no case for constructors"
> rename_e env ns (ECase e alts) = rename_case env ns e alts
> rename_case env ns e alts = error "rename_case: not yet written"
```

6.3.5 Collecting supercombinators

Finally, we have to name the supercombinators and collect them together. The main function, collectSCs_e, therefore has to return the collection of supercombinators it has found, as well as the transformed expression.

```
> collectSCs_e :: coreExpr -> ([coreScDefn], coreExpr)
```

collectSCs is defined using mapAccuml to do all the plumbing:

```
> collectSCs prog
> = concat (map collect_one_sc prog)
>   where
>   collect_one_sc (sc_name, args, rhs)
>   = (sc_name, args, rhs') : scs
>     where
>     (scs, rhs') = collectSCs_e rhs
```

The code for collectSCs_e is now easy to write. We just apply collectSCs_e recursively to the sub-expressions, collecting up the supercombinators thus produced.

```
> collectSCs_e (ENum k)      = ([], ENum k)
> collectSCs_e (EVar v)      = ([], EVar v)
> collectSCs_e (EAp e1 e2)   = (scs1 ++ scs2, EAp e1' e2')
>                              where
>                              (scs1, e1') = collectSCs_e e1
>                              (scs2, e2') = collectSCs_e e2


> collectSCs_e (ELam args body) = (scs, ELam args body')
>                                 where
>                                 (scs, body') = collectSCs_e body
> collectSCs_e (EConstr t a) = ([], EConstr t a)
> collectSCs_e (ECase e alts)
> = (scs_e ++ scs_alts, ECase e' alts')
>   where
>   (scs_e, e') = collectSCs_e e
>   (scs_alts, alts') = mapAccuml collectSCs_alt [] alts
>   collectSCs_alt scs (tag, args, rhs) = (scs++scs_rhs, (tag, args, rhs'))
>                                         where
>                                         (scs_rhs, rhs') = collectSCs_e rhs
```

The case for let(rec) is the interesting one. We need to process the definitions recursively and then split them into two groups: those of the form **v = \args. e** (the supercombinators), and the others (the non-supercombinators). The supercombinators are returned as part of the supercombinator list, and a new let(rec) is formed from the remaining non-supercombinators:

```
> collectSCs_e (ELet is_rec defns body)
```

```
> = (rhss_scs ++ body_scs ++ local_scs, mkELet is_rec non_scs' body')
>   where
>   (rhss_scs,defns') = mapAccuml collectSCs_d [] defns
>
>   scs'     = [(name,rhs) | (name,rhs) <- defns';   isELam rhs ]
>   non_scs' = [(name,rhs) | (name,rhs) <- defns'; ~(isELam rhs)]
>   local_scs = [(name,args,body) | (name,ELam args body) <- scs']
>
>   (body_scs, body') = collectSCs_e body
>
>   collectSCs_d scs (name,rhs) = (scs ++ rhs_scs, (name, rhs'))
>                                 where
>                                 (rhs_scs, rhs') = collectSCs_e rhs
```

The auxiliary function `isELam` tests for an `ELam` constructor; it is used to identify supercombinators.

```
> isELam :: expr * -> bool
> isELam (ELam args body) = True
> isELam other            = False
```

The `mkELet` function just builds an `ELet` expression:

```
> mkELet is_rec defns body = ELet is_rec defns body
```

6.4 Mark 2: Improving the simple lambda lifter

This completes the definition of the simple lambda lifter. We now consider some simple improvements.

6.4.1 Simple extensions

Exercise 6.3. The simple lambda lifter generates lots of `let` expressions with an empty list of bindings, because `collectSCs` removes the single binding from each of the supercombinator `let` expressions introduced by `abstract`. Modify `mkELet` to elide these redundant `let` expressions.

Exercise 6.4. Give definitions for `freeVars_case`, `abstract_case` and `collectSCs_case`, and test them.

6.4.2 Eliminating redundant supercombinators

Consider the Core-language program

```
f = \x. x+1
```

This will be transformed by `lambdaLift` to

```
f = $f
$f x = x+1
```

It would be nicer to avoid introducing the redundant definition. This improvement will become rather more significant when we come to consider full laziness, because many supercombinators of this form will be introduced.

Exercise 6.5. Add a special case to the function `collect_one_sc` (in `collectSCs`), to behave differently when `rhs` is a lambda abstraction. You should be able to avoid introducing a new supercombinator in this situation.

6.4.3 Eliminating redundant local definitions

A similar situation can arise with local definitions. Consider the Core-language program

```
f x = let g = (\y. y+1) in g (g x)
```

The lambda lifter will produce the program

```
f x = let g = $g in g (g x)
$g y = y+1
```

Exercise 6.6. Improve the definition of `collectSCs_d` (in the `ELet` case of `collectSCs_e`), so that it gives special treatment to definitions whose right-hand side is a lambda abstraction. For the above example you should generate

```
f x = g (g x)
g y = y+1
```

6.5 Mark 3: Johnsson-style lambda lifting

There is an interesting variant of the lambda lifting technique, which was discovered by [Johnsson 1985]. One slight problem with our current technique is that it produces programs in which many of the calls are to functions which are passed in as arguments. For example, consider the recursive example in Section 6.3:

```
f x = letrec g = \y. cons (x*y) (g y) in g 3
main = f 6
```

Our current lambda lifter produces the following set of supercombinators:

```
$g g x y = cons (x*y) (g y)
f x = letrec g = $g g x in g 3
main = f 6
```

Notice that `$g` makes a call to its argument `g`. In some implementations it would be more efficient if `$g` was directly recursive, like this:

```
$g x y = cons (x*y) ($g x y)
f x = $g x 3
main = f 6
```

The inner `letrec` has vanished altogether, and the supercombinator `g` has become directly recursive.

To get a more detailed idea of how to do Johnsson-style lambda lifting, we will look at a slightly more complicated example:

```
f x y = letrec
            g = \p. ...h...x...
            h = \q. ...g...y...
        in
        ...g...h...
```

Here, `g` is meant to be a function which calls `h`, and mentions the variable `x`; similarly `h` calls `g` and mentions `y`. The first step is to transform the definition like this:

```
f x y = letrec
            g = \x y p. ...(h x y)...x...
            h = \x y q. ...(g x y)...y...
        in
        ...(g x y)...(h x y)...
```

This transformation, which we call the *abstraction step*, is a little tricky. It does the following:

- take the free variables of the right-hand sides of the `letrec`, namely `g`, `h`, `x` and `y`;
- exclude the variables being bound (`g` and `h`) to leave just `x` and `y`;
- make these variables into extra arguments of each right-hand side;
- and replaced all occurrences of `g` with `(g x y)`, similarly for `h`.

It is important that we make `y` into an extra parameter of `g` even though `y` does not occur directly in its right-hand side, because `g` will need it to pass to `h`. In general, each member of the mutually recursive group must take as extra arguments the free variables of *all* the members together.

Now all we need do is float the definitions of `g` and `h` out to the top level, leaving:

```
f x y = ...(g x y)...(h x y)...
g x y p = ...(h x y)...x...
h x y q = ...(g x y)...y...
```

One last point. Before doing this process it is important that all binders are unique. Otherwise name clashes could arise, in two ways. The obvious way is that two supercombinators could have the same name. The less obvious way is illustrated by the following variant of the same example:

```
f x y = letrec
            g = \p. ...h...x...
            h = \x. ...g...y...
        in
        ...g...h...
```

Now `h` uses the same name for its argument as `f`, which will cause trouble when we try to make the free variable of `g`, namely `x`, into an extra argument to `h`! All in all, it is much easier simply to rename the program before starting work.

6.5.1 Implementation

The Johnsson-style lambda lifter can be implemented in several passes:

- The first pass renames the program so that all binders are unique. We can reuse the `rename` function for this purpose.
- Next, we annotate the program with its free-variable information, using the existing function `freeVars`.
- Now comes the main abstraction step discussed above:

  ```
  > abstractJ :: annProgram name (set name) -> coreProgram
  ```

- Finally, we can collect supercombinators with `collectSCs`.

The full Johnsson-style lambda lifter is just the composition of these stages:

```
> lambdaLiftJ = collectSCs . abstractJ . freeVars . rename
> runJ = pprint . lambdaLiftJ . parse
```

6.5.2 Abstracting free variables in functions

The only new function we need is `abstractJ`. The abstraction process makes substitutions as it goes along, replacing $\mathbf{g}$ with $\mathbf{g}\ \mathbf{v}_1 \ldots \mathbf{v_n}$, where $\mathbf{g}$ is one of the new supercombinators and $\mathbf{v}_1, \ldots, \mathbf{v_n}$ are the free variables of its declaration group. It follows that the auxiliary function `abstractJ_e` needs to take an environment mapping each supercombinator $\mathbf{g}$ to the free variables of its group $\mathbf{v}_1, \ldots, \mathbf{v_n}$:

```
> abstractJ_e :: assoc name [name]             || Maps each new SC to
>                                              ||   the free vars of its group
>                -> annExpr name (set name)    || Input expression
>                -> coreExpr                   || Result expression
```

To be fair, it looks as though the first argument could be of type `assoc name coreExpr` but, as we shall see, we need to make use of the environment in another way as well, which leads to the type we suggest here.

It is now easy to define `abstractJ` in terms of `abstractJ_e`, by applying the latter to each top-level definition:

```
> abstractJ prog = [ (name,args,abstractJ_e [] rhs)
>                  | (name, args, rhs) <- prog]
```

Now we come to `abstractJ_e`. The cases for constants and applications are easy.

```
> abstractJ_e env (free, ANum n)      = ENum n
> abstractJ_e env (free, AConstr t a) = EConstr t a
> abstractJ_e env (free, AAp e1 e2)   = EAp (abstractJ_e env e1)
>                                           (abstractJ_e env e2)
```

When we come to a variable $\mathbf{g}$, we look it up in the environment, getting back a list of variables $\mathbf{v}_1, \ldots, \mathbf{v_n}$. We then return the application $\mathbf{g}\ \mathbf{v}_1 \ldots \mathbf{v_n}$. If $\mathbf{g}$ does not appear in the environment we return the empty list from the environment lookup, and hence the 'application' we construct will simply be the variable $\mathbf{g}$!

```
> abstractJ_e env (free, AVar g)
> = foldll EAp (EVar g) (map EVar (aLookup env g []))
```

Sometimes we may find a lambda abstraction on its own; for example:

```
        f xs = map (\x. x+1) xs
```

The `\x`-abstraction is not the right-hand side of a `let(rec)` definition, so we treat it in just same way as we did in the simple lambda lifter (see the `ELam` case of `abstract`).

There is just one important difference. Since `abstractJ_e` is simultaneously performing a substitution on the expression, the free-variables information does not reflect the post-substitution state of affairs. Rather, we need to perform the substitution on the free-variable set too, to find what variables are free in the result. This is done by the function `actualFreeList`, which is defined at the end of this section. It was the need to perform this operation on the free-variable information which guided our choice of environment representation.

```
> abstractJ_e env (free, ALam args body)
> = foldll EAp sc (map EVar fv_list)
>   where
>   fv_list = actualFreeList env free
>   sc = ELet nonRecursive [("sc",sc_rhs)] (EVar "sc")
>   sc_rhs = ELam (fv_list ++ args) (abstractJ_e env body)
```

Lastly, we treat `let(rec)` expressions. Each variable bound to a lambda abstraction will be turned into a supercombinator, while the others (not bound to a lambda abstraction) will not. It follows that we need to treat separately these two kinds of definitions, which we call 'function definitions' and 'variable definitions' respectively.

```
> abstractJ_e env (free, ALet isrec defns body)
> = ELet isrec (fun_defns' ++ var_defns') body'
>   where
>   fun_defns = [(name,rhs) | (name,rhs) <- defns;   isALam rhs ]
>   var_defns = [(name,rhs) | (name,rhs) <- defns; ~(isALam rhs)]
```

Now that we have separated the function definitions from the variable definitions we can compute the set of variables to abstract from the functions. We take the union of the free variables of the function definitions, remove from this set the function names being bound, and then use `actualFreeList` (for the same reason as in the `ELam` equation) to get the result:

```
>   fun_names = bindersOf fun_defns
>   free_in_funs = setSubtraction
>                     (setUnionList [freeVarsOf rhs | (name,rhs)<-fun_defns])
>                     (setFromList fun_names)
>   vars_to_abstract = actualFreeList env free_in_funs
```

Next, we compute the new environment, to be used in the right-hand sides and in the body of the `let(rec)`:

```
>   body_env = [(fun_name, vars_to_abstract) | fun_name <- fun_names] ++ env
>   rhs_env = body_env, isrec
>           = env,      ~isrec
```

Lastly, we compute the new function definitions, variable definitions and body, by recursively using `abstractJ_E` with the appropriate environment:

```
>   fun_defns' = [ (name, ELam (vars_to_abstract ++ args)
>                              (abstractJ_e rhs_env body))
>                | (name, (free, ALam args body)) <- fun_defns
>                ]
>   var_defns' = [(name, abstractJ_e rhs_env rhs) | (name, rhs) <- var_defns]
>   body' = abstractJ_e body_env body
```

The function `actualFreeList` takes the environment and a set of free variables, applies the environment to the set, and returns a list (without duplicates) of the post-substitution free variables.

```
> actualFreeList :: assoc name [name] -> set name -> [name]
> actualFreeList env free
> = setToList (setUnionList [ setFromList (aLookup env name [name])
>                           | name <- setToList free
>                           ])
```

The function `isALam` identifies `ALam` constructors.

```
> isALam :: annExpr * ** -> bool
> isALam (free, ALam args body) = True
> isALam other                  = False
```

This concludes the Johnsson-style lambda lifter.

Exercise 6.7. Add a case for `case` expressions to the function `abstractJ_e`.

6.5.3 A tricky point†

When a `letrec` contains a mixture of function and variable definitions, the lambda lifter we have designed may introduce some redundant parameters. For example, consider the definition

```
f x y = letrec
          g = \p. h p + x ;
          h = \q. k + y + q;
          k = g y
        in
        g 4 ;
```

The free variables of the `g`/`h` group are `x`, `y` and `k`, so we will transform to:

```
f x y = letrec
           k = g x y k y
        in
        g 4

g x y k p = h x y k p + x ;
h x y k q = k + y + q;
```

Here the extra parameter `x` is not actually used in `h`, so better definitions for `g` and `h` would be

```
g x y k p = h y k p + x ;
h y k q = k + y + q;
```

Exercise 6.8. †Modify `abstractJ` to solve perform this more sophisticated transformation. Warning: this is quite a difficult job!

6.6 Mark 4: A separate full laziness pass

We now turn our attention to an important property of functional programs called *full laziness*. Previous accounts of full laziness have invariably linked it to lambda lifting, by describing 'fully lazy lambda lifting', which turns out to be rather a complex process. [Hughes 1983] gives an algorithm, but it is extremely subtle and does not handle `let(rec)` expressions. On the other hand, [Peyton Jones 1987] does cover `let(rec)` expressions, but the description is only informal and no algorithm is given.

In this section we show how full laziness and lambda lifting can be cleanly separated. This is done by means of a transformation involving `let` expressions. Lest it be supposed that we have simplified things in one way only by complicating them in another, we also show that performing fully lazy lambda lifting without `let(rec)` expressions risks an unexpected loss of laziness. Furthermore, much more efficient code can be generated for `let(rec)` expressions in later phases of most compilers than for their equivalent lambda expressions.

6.6.1 A review of full laziness

We begin by briefly reviewing the concept of full laziness. Consider again the example given in Section 6.3.

```
f x = let g = \y. x*x + y in (g 3 + g 4)
main = f 6
```

The simple lambda lifter generates the program:

```
$g x y = x*x + y
f x = let g = $g x in (g 3 + g 4)
main = f 6
```

In the body of `f` there are two calls to `g` and hence to `$g`. But `($g x)` is not a reducible expression, so `x*x` will be computed twice. But `x` is fixed in the body of `f`, so some work is being duplicated. It would be better to share the calculation of `x*x` between the two calls to `$g`. This can be achieved as follows: instead of making `x` a parameter to `$g`, we make `x*x` into a parameter, like this:

```
$g p y = p + y
f x = let g = $g (x*x) in (g 3 + g 4)
```

(we omit the definition of `main` from now on, since it does not change). So a fully lazy lambda lifter will make each *maximal free sub-expresssion* (rather than each free variable) of a lambda abstraction into an argument of the corresponding supercombinator. A maximal free expression (or MFE) of a lambda abstraction is an expression which contains no occurrences of the variable bound by the abstraction, and is not a sub-expression of a larger expression with this property.

Full laziness corresponds precisely to moving a loop-invariant expression outside the loop, so that it is computed just once at the beginning rather than once for each loop iteration.

How important is full laziness for 'real' programs? No serious studies have yet been made of this question, though we plan to do so. However, recent work by Holst suggests that the importance of full laziness may be greater than might at first be supposed [Holst 1990]. He shows how to perform a transformation which automatically enhances the effect of full laziness, to the point where the optimisations obtained compare favourably with those gained by partial evaluation [Jones *et al.* 1989], though with much less effort.

6.6.2 Fully-lazy lambda lifting in the presence of `let(rec)`s

Writing a fully lazy lambda lifter, as outlined in the previous section, is surprisingly difficult. Our language, which includes `let(rec)` expressions, appears to make this worse by introducing a new language construct. For example, suppose the definition of `g` in our running example was slightly more complex, thus:

```
g = \y. let z = x*x
        in let p = z*z
        in p + y
```

Now, the sub-expression `x*x` is an MFE of the `\y`-abstraction, but sub-expression `z*z` is not since `z` is bound inside the `\y`-abstraction. Yet it is clear that `p` depends only on `x` (albeit indirectly), and so we should ensure that `z*z` is only computed once.

Does a fully lazy lambda lifter spot this if `let` expressions are coded as lambda applications? No, it does not. The definition of `g` would become

```
g = \y. (\z. (\p. p+y) (z*z)) (x*x)
```

Now, `x*x` is free as before, but `z*z` is not. In other words, *if the compiler does not treat `let(rec)` expressions specially, it may lose full laziness which the programmer might reasonably expect to be preserved.*

Fortunately, there is a straightforward way to handle `let(rec)` expressions – described in [Peyton Jones 1987, Chapter 15] – namely to 'float' each `let(rec)` definition outward until it is outside any lambda abstraction in which it is free. For example, all we need do is transform the definition of `g` to the following:

```
g = let z = x*x
    in let p = z*z
    in \y. p + y
```

Now `x*x` and `z*z` will each be computed only once. Notice that this property should hold *for any implementation of the language*, not merely for one based on lambda lifting and graph reduction. This is a clue that full laziness and lambda lifting are not as closely related as at first appears, a topic to which we will return in the next section.

Meanwhile, how can we decide how far out to float a definition? It is most easily done by using *lexical level numbers* (or *de Bruijn numbers*). There are three steps:

- First, assign to each lambda-bound variable a level number, which says how many lambdas enclose it. Thus in our example, `x` would be assigned level number 1, and `y` level number 2.
- Now, assign a level number to each `let(rec)`-bound variable (outermost first), which is the maximum of the level numbers of its free variables, or zero if there are none. In our example, both `p` and `z` would be assigned level number 1. Some care needs to be taken to handle `letrec`s correctly.
- Finally, float each definition (whose binder has level $\mathbf{n}$, say) outward, until it is outside the lambda abstraction whose binder has level $\mathbf{n}+1$, but still inside the level-$\mathbf{n}$ abstraction. There is some freedom in this step about exactly where between the two the definition should be placed.

Each mutually recursive set of definitions defined in a `letrec` should be floated out together, because they depend on each other and must remain in a single `letrec`. If, in fact, the definitions are *not* mutually recursive despite appearing in the same `letrec`, this policy might lose laziness by retaining in an inner scope a definition which could otherwise be floated further outwards. The standard solution is to perform *dependency analysis* on the definitions in each `letrec` expression, to break each group of definitions into its minimal subgroups. We will look at this in Section 6.8.

Finally, a renaming pass should be carried out before the floating operation, so that there is no risk that the bindings will be altered by the movement of the `let(rec)` definitions. For example, the expression

```
\y. let y = x*x in y
```

is obviously not equivalent to

```
let y = x*x in \y->y
```

All that is required is to give every binder a unique name to eliminate the name clash.

6.6.3 Full laziness without lambda lifting

At first it appears that the requirement to float `let(rec)`s outward in order to preserve full laziness merely further complicates the already subtle fully lazy lambda lifting algorithm suggested by Hughes. However, a simple transformation allows *all* the full laziness to be achieved by `let(rec)` floating, while lambda lifting is performed by the original simple lambda lifter.

The transformation is this: *before floating `let(rec)` definitions, replace each MFE* `e` *with the expression `let v = e in v`.* This transformation both gives a name to the MFE and makes it accessible to the `let(rec)` floating transformation, which can now float out the new definitions. Ordinary lambda lifting can then be performed. For example, consider the original definition of `g`:

```
f x = let g = \y. x*x + y
      in (g 3 + g 4)
main = f 6
```

The sub-expression `x*x` is an MFE, so it is replaced by a trivial `let` expression:

```
f x = let g = \y. (let v = x*x in v) + y
      in (g 3 + g 4)
main = f 6
```

Now the `let` expression is floated outward:

```
f x = let g = let v = x*x in \y. v + y
      in (g 3 + g 4)
in
f 6
```

Finally, ordinary lambda lifting will discover that v is free in the \y-abstraction, and the resulting program becomes:

```
$g v y = v + y
f x = let g = let v = x*x in $g v
      in (g 3 + g 4)
main = f 6
```

A few points should be noted here. Firstly, the original definition of a maximal free expression was relative to a *particular* lambda abstraction. The new algorithm we have just developed transforms certain expressions into trivial `let` expressions. Which expressions are so transformed? Just the ones which are MFEs of *any* enclosing lambda abstraction. For example, in the expression:

```
\y. \z. (y + (x*x)) / z
```

two MFEs are identified: `(x*x)`, since it is an MFE of the \y-abstraction, and `(y + (x*x))`, since it is an MFE of the \z-abstraction. After introducing the trivial `let` bindings, the expression becomes

```
\y. \z. (let v1 = y + (let v2 = x*x in v2) in v1) / z
```

Secondly, the newly introduced variable v must either be unique, or the expression must be uniquely renamed after the MFE-identification pass.

Thirdly, in the final form of the program v is only referenced once, so it would be sensible to replace the reference by the right-hand side of the definition and eliminate the definition, yielding exactly the program we obtained using Hughes's algorithm. This is a straightforward transformation, and we will not discuss it further here, except to note that this property will hold for all `let` definitions which are floated out past a lambda. In any case, many compiler back ends will generate the same code regardless of whether or not the transformation is performed.

6.6.4 A fully lazy lambda lifter

Now we are ready to define the fully lazy lambda lifter. It can be decomposed into the following stages:

- First we must make sure that each `ELam` constructor and supercombinator definition binds only a single argument, because the fully lazy lambda lifter must treat each lambda individually. It would be possible to encode this in later phases of the algorithm, by dealing with a list of arguments, but it turns out that we can express an important optimisation by altering this pass alone.

  ```
  > separateLams :: coreProgram -> coreProgram
  ```

- First we annotate all binders and expressions with level numbers, which we represent by natural numbers starting with zero:

```
> level == num
> addLevels :: coreProgram -> annProgram (name, level) level
```

- Next we identify all MFEs, by replacing them with trivial `let` expressions. Level numbers are no longer required on every sub-expression, only on binders.

```
> identifyMFEs :: annProgram (name, level) level -> program (name, level)
```

- A renaming pass makes all binders unique, so that floating does not cause name-capture errors. This must be done after `identifyMFEs`, which introduces new bindings. Sadly, this means that we cannot use our current `rename` function because it works on a `coreProgram`, whereas `identifyMFEs` has produced a `program (name, level)`. We invent a new function `renameL` for the purpose:

```
> renameL ::  program (name, *) -> program (name, *)
```

- Now the `let(rec)` definitions can be floated outwards. The level numbers are not required any further.

```
> float :: program (name,level) -> coreProgram
```

- Finally, ordinary lambda lifting can be carried out, using `lambdaLift` from Section 6.3.1.

The fully lazy lambda lifter is just the composition of these passes:

```
> fullyLazyLift = float . renameL . identifyMFEs . addLevels . separateLams
> runF          = pprint . lambdaLift . fullyLazyLift . parse
```

As before, we leave most of the equations for `case` expressions as an exercise.

6.6.5 Separating the lambdas

We define `separateLams` in terms of an auxiliary function `separateLams_e`, which recursively separates variables bound in lambda abstractions in expressions:

```
> separateLams_e :: coreExpr -> coreExpr
> separateLams_e (EVar v) = EVar v
> separateLams_e (EConstr t a) = EConstr t a
> separateLams_e (ENum n) = ENum n
> separateLams_e (EAp e1 e2) = EAp (separateLams_e e1) (separateLams_e e2)
> separateLams_e (ECase e alts)
> = ECase (separateLams_e e) [ (tag, args, separateLams_e e)
>                            | (tag, args, e) <- alts
>                            ]
>
> separateLams_e (ELam args body) = mkSepLams args (separateLams_e body)
```

```
>
> separateLams_e (ELet is_rec defns body)
> = ELet is_rec [(name, separateLams_e rhs) | (name,rhs) <- defns]
>               (separateLams_e body)

> mkSepLams args body = foldr mkSepLam body args
>                       where mkSepLam arg body = ELam [arg] body
```

Now we return to the top-level function separateLams. The interesting question is what to do about supercombinator definitions. The easiest thing to do is to turn them into the equivalent lambda abstractions!

```
> separateLams prog = [ (name, [], mkSepLams args (separateLams_e rhs))
>                     | (name, args, rhs) <- prog
>                     ]
```

6.6.6 Adding level numbers

There are a couple of complications concerning annotating an expression with level numbers. At first it looks as though it is sufficient to write a function which returns an expression annotated with level numbers; then for an application, for example, one simply takes the maximum of the levels of the two sub-expressions. Unfortunately, this approach loses too much information, because there is no way of mapping the level number of the *body* of a lambda abstraction to the level number of the abstraction *itself*. The easiest solution is first to annotate the expression with its free variables, and then use a mapping freeSetToLevel from variables to level numbers, to convert the free-variable annotations to level numbers.

```
> freeSetToLevel :: assoc name level -> set name -> level
> freeSetToLevel env free
> = max (0:[aLookup env n 0 | n <- setToList free])
>   || If there are no free variables, return level zero
```

The second complication concerns letrec expressions. What is the correct level number to attribute to the newly introduced variables? The right thing to do is to take the maximum of the levels of the free variables of all the right-hand sides *without* the recursive variables, or equivalently map the recursive variables to level zero when taking this maximum. This level should be attributed to each of the new variables. let expressions are much simpler: just attribute to each new variable the level number of its right-hand side.

Now we are ready to define addLevels. It is the composition of two passes, the first of which annotates the expression with its free variables, while the second uses this information to generate level-number annotations.

```
> addLevels = freeToLevel . freeVars
```

We have defined the freeVars function already, so it remains to define freeToLevel. The main function will need to carry around the current level, and a mapping from variables to level numbers, so as usual we define freeToLevel in terms of freeToLevel_e which does all the work.

```
> freeToLevel_e :: level                     -> || Level of context
>                  assoc name level          -> || Level of in-scope names
>                  annExpr name (set name)   -> || Input expression
>                  annExpr (name, level) level || Result expression
```

We represent the name-to-level mapping as an association list, with type `assoc name level`. The interface of association lists is given in Appendix A, but notice that it is *not* abstract. It is so convenient to use all the standard functions on lists, and notation for lists, rather than to invent their analogues for associations, that we have compromised the abstraction.

Now we can define `freeToLevel`, using an auxiliary function to process each supercombinator definition. Remember that `separateLams` has removed all the arguments from supercombinator definitions:

```
> freeToLevel prog = map freeToLevel_sc prog
>
> freeToLevel_sc (sc_name, [], rhs) = (sc_name, [], freeToLevel_e 0 [] rhs)
```

For constants, variables and applications, it is simpler and more efficient to ignore the free-variable information and calculate the level number directly.

```
> freeToLevel_e level env (free, ANum k) = (0, ANum k)
> freeToLevel_e level env (free, AVar v) = (aLookup env v 0, AVar v)


> freeToLevel_e level env (free, AConstr t a) = (0, AConstr t a)


> freeToLevel_e level env (free, AAp e1 e2)
> = (max [levelOf e1', levelOf e2'], AAp e1' e2')
>    where
>    e1' = freeToLevel_e level env e1
>    e2' = freeToLevel_e level env e2
```

The same cannot be done for lambda abstractions; so we must compute the level number of the abstraction using `freeSetToLevel`. We also assign a level number to each variable in the argument list. At present we expect there to be only one such variable, but we will allow there to be several and assign them all the same level number. This works correctly now, and turns out to be just what is needed to support a useful optimisation later (Section 6.7.3).

```
> freeToLevel_e level env (free, ALam args body)
> = (freeSetToLevel env free, ALam args' body')
>   where
>   body' = freeToLevel_e (level + 1) (args' ++ env) body
>   args' = [(arg, level+1) | arg <- args]
```

let(rec) expressions follow the scheme outlined at the beginning of this section.

```
> freeToLevel_e level env (free, ALet is_rec defns body)
> = (levelOf new_body, ALet is_rec new_defns new_body)
>   where
>   binders = bindersOf defns
>   rhss = rhssOf defns
>
>   new_binders = [(name,max_rhs_level) | name <- binders]
>   new_rhss = map (freeToLevel_e level rhs_env) rhss
>   new_defns = zip2 new_binders new_rhss
>   new_body = freeToLevel_e level body_env body
>
>   free_in_rhss = setUnionList [free | (free,rhs) <- rhss]
>   max_rhs_level = freeSetToLevel level_rhs_env free_in_rhss
>
>   body_env      = new_binders ++ env
>   rhs_env       = body_env,                                 is_rec
>                 = env,                                      otherwise
>   level_rhs_env = [(name,0) | name <- binders] ++ env,      is_rec
>                 = env,                                      otherwise
```

Notice that the level of the whole let(rec) expression is that of the body. This is valid provided the body refers to all the binders directly or indirectly. If any definition is unused, we might assign a level number to the letrec which would cause it to be floated outside the scope of some variable mentioned in the unused definition. This is easily fixed, but it is simpler to assume that the expression contains no redundant definitions; the dependency analysis which we look at in the next section will eliminate such definitions.

case expressions are deferred:

```
> freeToLevel_e level env (free, ACase e alts)
> = freeToLevel_case level env free e alts
> freeToLevel_case free e alts = error "freeToLevel_case: not yet written"
```

Lastly the auxiliary functions levelOf extracts the level from an expression:

```
> levelOf :: annExpr * level -> level
> levelOf (level, e) = level
```

6.6.7 Identifying MFEs

It is simple to identify MFEs, by comparing the level number of an expression with the level of its context. This requires an auxiliary parameter to give the level of the context.

```
> identifyMFEs_e :: level                             || Level of context
>                   -> annExpr (name, level) level    || Input expression
>                   -> expr (name, level)             || Result
```

```
> identifyMFEs prog = [ (sc_name, [], identifyMFEs_e 0 rhs)
>                     | (sc_name, [], rhs) <- prog
>                     ]
```

Once an MFE `e` has been identified, our strategy is to wrap it in a trivial `let` expression of the form `let v = e in v`; but not all MFEs deserve special treatment in this way. For example, it would be a waste of time to wrap such a `let` expression around an MFE consisting of a single variable or constant. Other examples are given in Section 6.7.3. We encode this knowledge of which MFEs deserve special treatment in a function `notMFECandidate`.

```
> notMFECandidate (AConstr t a) = True
> notMFECandidate (ANum k)      = True
> notMFECandidate (AVar v)      = True
> notMFECandidate ae            = False || For now everything else
>                                       ||         is a candidate
```

`identifyMFEs_e` works by comparing the level number of the expression with that of its context. If they are the same, or for some other reason the expression is not a candidate for special treatment, the expression is left unchanged, except that `identifyMFEs_e1` is used to apply `identifyMFEs_e` to its sub-expressions; otherwise we use `transformMFE` to perform the appropriate transformation.

```
> identifyMFEs_e cxt (level, e)
> = e',                         level = cxt \/ notMFECandidate e
> = transformMFE level e',      otherwise
>   where
>   e' = identifyMFEs_e1 level e

> transformMFE level e = ELet nonRecursive [(("v",level), e)] (EVar "v")
```

`identifyMFEs_e1` applies `identifyMFEs_e` to the components of the expression.

```
> identifyMFEs_e1 :: level                                || Level of context
>                 -> annExpr' (name,level) level          || Input expressions
>                 -> expr (name,level)                    || Result expression

> identifyMFEs_e1 level (AConstr t a)          = EConstr t a
> identifyMFEs_e1 level (ANum n)               = ENum n
> identifyMFEs_e1 level (AVar v)               = EVar v
> identifyMFEs_e1 level (AAp e1 e2)
> = EAp (identifyMFEs_e level e1) (identifyMFEs_e level e2)
```

When `identifyMFEs_e1` encounters a binder it changes the 'current' level number carried down as its first argument, as we can see in the equations for lambda abstractions and `let(rec)` expressions:

```
> identifyMFEs_e1 level (ALam args body)
> = ELam args (identifyMFEs_e arg_level body)
>   where
>   (name, arg_level) = hd args

> identifyMFEs_e1 level (ALet is_rec defns body)
> = ELet is_rec defns' body'
>   where
>   body' = identifyMFEs_e level body
>   defns' = [ ((name, rhs_level), identifyMFEs_e rhs_level rhs)
>            | ((name, rhs_level), rhs) <- defns
>            ]
```

`case` expressions are deferred:

```
> identifyMFEs_e1 level (ACase e alts) = identifyMFEs_case1 level e alts
> identifyMFEs_case1 level e alts = error "identifyMFEs_case1: not written"
```

6.6.8 Renaming variables

As we remarked above, it would be nice to use the existing `rename` function to make the binders unique, but it has the wrong type. It would be possible to write `renameL` by making a copy of `rename` and making some small alterations, but it would be much nicer to make a single generic renaming function, `renameGen`, which can be specialised to do either `rename` or `renameL`.

What should the type of `renameGen` be? The right question to ask is: *'what use did we make in `rename` of the fact that each binder was a simple `name`?'* or, alternatively, 'what operations did we perform on binders in `rename`?'.

There is actually just one such operation, which constructs new binders. In `rename_e` this function is called `newNames`; it takes a name supply and a list of names, and returns a depleted name supply, a list of new names and an association list mapping old names to new ones:

```
> newNames :: nameSupply -> [name]  -> (nameSupply, [name], assoc name name)
```

Since `renameGen` must be able to work over any kind of binder, not just those of type `name`, *we must pass the new-binders function into `renameGen` as an extra argument.* So the type of `renameGen` is:

```
> renameGen :: (nameSupply -> [*]  -> (nameSupply, [*], assoc name name))
>                                   || New-binders function
>              -> program *         || Program to be renamed
>              -> program *         || Resulting program
```

Notice that the type of the binders is denoted by the type variable `*`, because `renameGen` is polymorphic in this type. Using `renameGen`, we can now redefine the original `rename` function, by passing `newNames` to `renameGen` as the new-binders function.

```
> rename :: CoreProgram -> CoreProgram
> rename prog = renameGen newNames prog
```

renameL is rather more interesting. Its binders are (name,level) pairs so we need to define a different new-binders function:

```
> renameL :: program (name,level) -> program (name,level)
> renameL prog = renameGen newNamesL prog
```

The function newNamesL does just what newNames does, but it does it for binders whose type is a (name,level) pair:

```
> newNamesL ns old_binders
> = (ns', new_binders, env)
>   where
>   old_names = [name | (name,level) <- old_binders]
>   levels       = [level | (name,level) <- old_binders]
>   (ns', new_names) = getNames ns old_names
>   new_binders =  zip2 new_names levels
>   env = zip2 old_names new_names
```

Now we can turn our attention to writing renameGen. As usual we need an auxiliary function renameGen_e which carries around some extra administrative information. Specifically, like rename_e, it needs to take a name supply and old-name to new-name mapping as arguments, and return a depleted supply as part of its result. It also needs to be passed the new-binders function:

```
> renameGen_e :: (nameSupply -> [*]  -> (nameSupply, [*], assoc name name))
>                                                      || New-binders function
>                -> assoc name name                    || Maps old names to new ones
>                -> nameSupply                         || Name supply
>                -> expr *                             || Expression to be renamed
>                -> (nameSupply, expr *)               || Depleted name supply
>                                                      || and result expression
```

Using renameGen_e we can now write renameGen. Just like rename, renameGen applies a local function rename_sc to each supercombinator definition.

```
> renameGen new_binders prog
> = second (mapAccuml rename_sc initialNameSupply prog)
>    where
>    rename_sc ns (sc_name, args, rhs)
>    = (ns2, (sc_name, args', rhs'))
>       where
>      (ns1, args', env) = new_binders ns args
>      (ns2, rhs') = renameGen_e new_binders env ns1 rhs
```

Exercise 6.9. Write the function `renameGen_e`. It is very like `rename_e`, except that it takes the binder-manipulation functions as extra arguments. In the equations for `ELet`, `ELam` and `ECase` (which each bind new variables), the function `newBinders` can be used in just the same way as it is in `rename_sc` above.

Test your definition by checking that the simple lambda lifter still works with the new definition of `rename`.

Exercise 6.10. The type signature we wrote for `renameL` is actually slightly more restrictive than it need be. How could it be made more general (without changing the code at all)? Hint: what use does `renameL` make of the fact that the second component of a binder is of type `level`?

This section provides a good illustration of the way in which higher-order functions can help us to make programs more modular.

6.6.9 Floating `let(rec)` expressions

The final pass floats `let(rec)` expressions out to the appropriate level. The auxiliary function, which works over expressions, has to return an expression together with the collection of definitions which should be floated outside the expression.

```
> float_e :: expr (name, level) -> (floatedDefns, expr name)
```

There are many possible representations for the `floatedDefns` type, and we will choose a simple one, by representing the definitions being floated as a list, each element of which represents a group of definitions, identified by its level, and together with its `isRec` flag.

```
> floatedDefns == [(level, isRec, [(name, expr name)])]
```

Since the definitions in the list may depend on one another, we add the following constraint:

> *a definition group may depend only on definition groups appearing* earlier *in the* `floatedDefns` *list.*

We can now proceed to a definition of `float_e`. The cases for variables, constants and applications are straightforward.

```
> float_e (EVar v) = ([], EVar v)
> float_e (EConstr t a) = ([], EConstr t a)
> float_e (ENum n) = ([], ENum n)
> float_e (EAp e1 e2) = (fd1 ++ fd2, EAp e1' e2')
>                       where
>                       (fd1, e1') = float_e e1
>                       (fd2, e2') = float_e e2
```

How far out should a definition be floated? There is more than one possible choice, but here we choose to install a definition just inside the innermost lambda which binds one its free variables (recall from Section 6.6.6 that all variables bound by a single `ELam` construct are given the same level):

```
> float_e (ELam args body)
> = (fd_outer, ELam args' (install fd_this_level body'))
>   where
>   args' = [arg | (arg,level) <- args]
>   (first_arg,this_level) = hd args
>   (fd_body, body') = float_e body
>   (fd_outer, fd_this_level) = partitionFloats this_level fd_body
```

The equation for a let(rec) expression adds its definition group to those floated out from its body, and from its right-hand sides. The latter must come first, since the new definition group may depend on them.

```
> float_e (ELet is_rec defns body)
> = (rhsFloatDefns ++ [thisGroup] ++ bodyFloatDefns, body')
>   where
>   (bodyFloatDefns, body') = float_e body
>   (rhsFloatDefns, defns') = mapAccuml float_defn [] defns
>   thisGroup = (thisLevel, is_rec, defns')
>   (name,thisLevel) = hd (bindersOf defns)
>
>   float_defn floatedDefns ((name,level), rhs)
>   = (rhsFloatDefns ++ floatedDefns, (name, rhs'))
>     where
>     (rhsFloatDefns, rhs') = float_e rhs
```

We defer case expressions:

```
> float_e (ECase e alts) = float_case e alts
> float_case e alts = error "float_case: not yet written"
```

The auxiliary function partitionFloats takes a floatedDefns and a level number, and separates it into two: those belonging to an outer level and those belonging to the specified level (or an inner one):

```
> partitionFloats :: level -> floatedDefns -> (floatedDefns, floatedDefns)
> partitionFloats this_level fds
> = (filter is_outer_level fds, filter is_this_level fds)
>   where
>   is_this_level  (level,is_rec,defns) = level >= this_level
>   is_outer_level (level,is_rec,defns) = level <  this_level
```

The function install wraps an expression in a nested set of let(rec)s containing the specified definitions:

```
> install :: floatedDefns -> expr name -> expr name
> install defnGroups e
> = foldr installGroup e defnGroups
>   where
>   installGroup (level, is_rec, defns) e = ELet is_rec defns e
```

Finally, we can define the top-level function, `float`. It uses `float_sc` to apply `float_e` to each supercombinator, yielding a list of supercombinators, in just the same way as `collectSCs` above.

```
> float prog = concat (map float_sc prog)
```

The function `float_sc` takes a supercombinator definition to a list of supercombinator definitions, consisting of the transformed version of the original definition together with the level-zero definitions floated out from its body:

```
> float_sc (name, [], rhs)
> = [(name, [], rhs')] ++ concat (map to_scs fds)
>   where
>   (fds, rhs') = float_e rhs
>   to_scs (level, is_rec, defns) = map make_sc defns
>   make_sc (name, rhs) = (name, [], rhs)
```

The top level of a program is implicitly mutually recursive, so we can drop the `isRec` flags. We also have to give each floated definition an empty argument list, since it is now a supercombinator definition.

6.7 Mark 5: Improvements to full laziness

That completes the definition of the fully lazy lambda lifter. Its output is always correct, but it is larger and less efficient than it need be. In this section we discuss some ways to improve the full laziness transformation.

6.7.1 Adding `case` expressions

Exercise 6.11. Write definitions for `freeToLevel_case`, `identifyMFEs_case1` and `float_case`. All of them work in an analogous way to lambda abstractions. Hint: in `float_case` take care with alternatives whose argument list is empty.

6.7.2 Eliminating redundant supercombinators

Consider the Core-language expression

```
\x. \y. x+y
```

Here the `\y`-abstraction has no MFEs apart from `x` itself, so the full-laziness pass will not affect the expression at all. Unfortunately, the simple lambda lifter, `lambdaLift`, will then generate *two* supercombinators, one for each lambda, whereas only one is needed. It would be better to combine nested `ELam` expressions into a single `ELam` before passing the program to `lambdaLift`, so that the latter would then generate just one supercombinator. We could do this in a separate pass, but it saves work to do it as part of the work of `float`.

Exercise 6.12. Modify the definition of the `ELam` case of `float` so that it combines nested `ELam` constructors. Hint: make use of the function:

```
> mkELam :: [name] -> coreExpr -> coreExpr
> mkELam args (ELam args' body) = ELam (args++args') body
> mkELam args other_body        = ELam args          other_body
```

6.7.3 Avoiding redundant full laziness

Full laziness does not come for free. It has two main negative effects:

- Multiple lambda abstractions, such as `\x y. E`, turn into one supercombinator under the simple scheme, but may turn into two under the fully lazy scheme. Two reductions instead of one are therefore required to apply it to two arguments, which may well be more expensive.
- Lifting out MFEs removes sub-expressions from their context, and thereby reduces opportunities for a compiler to perform optimisations. Such optimisations might be partially restored by an interprocedural analysis which figured out the contexts again, but it is better still to avoid creating the problem.

These points are elaborated by [Fairbairn 1985] and [Goldberg 1988]. Furthermore, they point out that often no benefit arises from lifting out *every* MFE from *every* lambda abstraction. In particular:

- If no partial applications of a multiple abstraction can be shared, then nothing is gained by floating MFEs out to *between* the nested abstractions.
- Very little is gained by lifting out an MFE that is not a reducible expression. No work is shared thereby, though there may be some saving in storage because the closure need only be constructed once. This is more than outweighed by the loss of compiler optimisations caused by removing the expression from its context.

These observations suggest some improvements to the fully lazy lambda lifter, and they turn out to be quite easy to incorporate:

- If a multiple abstraction is *not* separated into separate `ELam` constructors by the `separateLam` pass, then all the variables bound by it will be given the *same* level number. It follows that no MFE will be identified which is free in the inner abstraction but not the outer one. This ensures that no MFEs will be floated out to between two abstractions represented by a single `ELam` constructor.

  All that is required is to modify the `separateLams` pass to keep in a single `ELam` constructor each multiple abstraction of which partial applications cannot be shared. This sharing information is not trivial to deduce, but at least we have an elegant way to use its results by modifying only a small part of our algorithm.

  This is one reason why we allow `ELam` constructors to take a list of binders.

- `identifyMFEs` uses a predicate `notMFECandidate` to decide whether to identify a particular sub-expression as an MFE. This provides a convenient place to add extra conditions to exclude from consideration expressions which are not redexes. This condition, too, is undecidable in general, but a good approximation can be made in many cases; for example `(+ 3)` is obviously not a redex.

This concludes the presentation of the full laziness transformation.

6.8 Mark 6: Dependency analysis†

Consider the Core-language definition

```
f x = let
        g = \y. letrec
                  h = y+1 ;
                  k = x+2
                in
                h+k
      in
      g 4
```

The inner `letrec` is not recursive at all! The program is equivalent to the following:

```
f x = let
        g = \y. let h = y+1 in
                let k = x+2 in
                h+k
      in
      g 4
```

This transformation, which breaks up `let(rec)` blocks into minimal-sized groups, and always uses `let` in preference to `letrec`, is called *dependency analysis*.

We have already alluded to the fact that better code can be generated if the program has been subjected to a dependency analysis. This has shown up in two places:

- In Johnsson-style lambda lifting, we treated the free variables for a `letrec` block of definitions as a single entity. If we could in fact break the `letrec` up into smaller blocks, then there would be fewer free variables in each block. This will then reduce the number of free variables that must be added to a function definition by the `abstractJ` function.
- In the full-laziness transformation we always kept the declarations in a `let(rec)` block together. If we first do dependency analysis, to break the declarations into small groups, then perhaps some of the groups could be floated further out than before.

This section explores how to do dependency analysis on Core programs.

6.8.1 Strongly connected components

In order to discuss the dependency analysis we need to understand some graph theory. We therefore begin with a definition.

Definition 6.1 *A directed graph is a tuple* $(\mathbf{V}, \mathbf{E})$ *of two components:*

- *a set of vertices (or nodes),* $\mathbf{V}$*;*
- *a set of edges,* $\mathbf{E}$*. Each edge is a pair:* $(\mathbf{v}_0, \mathbf{v}_1)$*; where* $\mathbf{v}_0 \in \mathbf{V}$ *is the source of the edge, and* $\mathbf{v}_1 \in \mathbf{V}$ *is the target.*

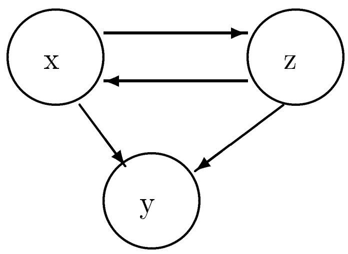
In the following expression we say that `x` depends on `y` and `z`, `z` depends on `x` and `y`, but that `y` does not depend on any other variable.

```
letrec
    x = y + 7 * tl z;
    y = 5
    z = (x,y)
in e
```

The graph we construct for this definition block has three vertices, $\{\mathtt{x}, \mathtt{y}, \mathtt{z}\}$, and the edges:

$$\{(\mathtt{x}, \mathtt{y}),\ (\mathtt{x}, \mathtt{z}),\ (\mathtt{z}, \mathtt{x}),\ (\mathtt{z}, \mathtt{y})\}$$

The interpretation of the first edge is that `x` *depends* on `y`, and the absence of any edges from `y` means that `y` does not depend on any other variables. Pictorially, the graph is as follows:



Because we are not concerned with multiple edges between the same pairs of vertices, we can instead formulate the information about the edges as a map: *outs*.

Definition 6.2

$$\mathit{outs}\ \mathbf{v} = \{\mathbf{v}' \mid (\mathbf{v}, \mathbf{v}') \in \mathbf{E}\}$$

The set *outs* $\mathbf{v}$ is therefore the set of vertices that are targets for an edge whose source is $\mathbf{v}$. In the example we have just looked at $\mathit{outs}\ \mathtt{x} = \{\mathtt{y}, \mathtt{z}\}$, $\mathit{outs}\ \mathtt{y} = \{\}$ and $\mathit{outs}\ \mathtt{z} = \{\mathtt{x}, \mathtt{y}\}$.

We can construct a similar map, *ins*, which is dual to the *outs* map.

Definition 6.3

$$\mathit{ins}\ \mathbf{v} = \{\mathbf{v}' \mid (\mathbf{v}', \mathbf{v}) \in \mathbf{E}\}$$

This is the set of vertices that are the source of an edge pointing to $\mathbf{v}$. In the example we have just looked at $\mathit{ins}\ \mathtt{x} = \{\mathtt{z}\}$, $\mathit{ins}\mathtt{y} = \{\mathtt{x}, \mathtt{z}\}$ and $\mathit{ins}\mathtt{z} = \{\mathtt{x}\}$.

Definition 6.4 *The map* $\mathbf{r}^*$ *is the* transitive closure *of the map* $\mathbf{r}$*. It can be defined recursively as:*

- $\mathbf{a} \in \mathbf{r}^*\ \mathbf{a}$,
- *if* $\mathbf{b} \in \mathbf{r}^*\ \mathbf{a}$, *then* $\mathbf{r}\ \mathbf{b} \subseteq \mathbf{r}^*\ \mathbf{a}$.

The set $\mathit{outs}^*\ \mathbf{a}$ is the set of all vertices that can be reached from the vertex $\mathbf{a}$, by following the edges of the graph. We say that $\mathbf{b}$ is *reachable* from $\mathbf{a}$ whenever $\mathbf{b} \in \mathit{outs}^*\ \mathbf{a}$. The set $\mathit{ins}^*\ \mathbf{a}$ is the set of all vertices that can reach the vertex $\mathbf{a}$ by following the edges of the graph.

Using the running example, we have $\mathit{outs}^*\ \mathtt{x} = \{\mathtt{x}, \mathtt{y}, \mathtt{z}\}$, $\mathit{outs}^*\ \mathtt{y} = \{\mathtt{y}\}$, and $\mathit{outs}^*\ \mathtt{z} = \{\mathtt{x}, \mathtt{y}, \mathtt{z}\}$. We also have $\mathit{ins}^*\ \mathtt{x} = \{\mathtt{x}, \mathtt{z}\}$, $\mathit{ins}^*\ \mathtt{y} = \{\mathtt{x}, \mathtt{y}, \mathtt{z}\}$, and $\mathit{ins}^*\ \mathtt{z} = \{\mathtt{x}, \mathtt{z}\}$.

We are now in a position to define a strongly connected component of a graph. Informally, the vertices $\mathbf{a}$ and $\mathbf{b}$ are in the same component whenever $\mathbf{a}$ is reachable from $\mathbf{b}$ and $\mathbf{b}$ is reachable from $\mathbf{a}$.

Definition 6.5 *The strongly connected component of the graph containing the vertex* $\mathbf{a}$ *is the set of vertices* $\mathit{scc}\ \mathbf{a}$*, defined as:*

$$\mathit{scc}\ \mathbf{a} = \mathit{outs}^*\ \mathbf{a} \cap \mathit{ins}^*\ \mathbf{a}$$

That is: a vertex $\mathbf{b}$ is in $\mathit{scc}\ \mathbf{a}$ if and only if it is reachable from $\mathbf{a}$ and if $\mathbf{a}$ is reachable from $\mathbf{b}$. The graph in the running example has two strongly connected components: $\{\mathtt{x}, \mathtt{z}\}$ and $\{\mathtt{y}\}$.

Exercise 6.13. Prove that the relation 'in the same strongly connected component' partitions the set of vertices into equivalence classes.

Topological sorting

An ordering on vertices can be induced by considering the maps ins^* and outs^*. This is a partial order which we will represent as $\preceq$.

Definition 6.6 *The vertex* $\mathbf{a}$ *is topologically less than or equal to the vertex* $\mathbf{b}$*, written* $\mathbf{a} \preceq \mathbf{b}$*, if and only if:*

$$\mathbf{b} \in \mathit{outs}^*\ \mathbf{a}$$

In our example $\mathtt{x} \preceq \mathtt{x}$, $\mathtt{x} \preceq \mathtt{y}$, $\mathtt{x} \preceq \mathtt{z}$, $\mathtt{y} \preceq \mathtt{y}$, $\mathtt{z} \preceq \mathtt{x}$, $\mathtt{z} \preceq \mathtt{y}$ and $\mathtt{z} \preceq \mathtt{z}$.

Because the strongly connected components are disjoint, a similar ordering is induced on them by considering the action of $\preceq$ on their elements. In our running example the two components are ordered as: $\{\mathtt{x}, \mathtt{z}\} \preceq \{\mathtt{y}\}$.

Definition 6.7 *A sequence of vertices (or strongly connected components) are topologically sorted if, for all* $\mathbf{a}$ *and* $\mathbf{b}$ *in the sequence,* $\mathbf{a}$ *precedes* $\mathbf{b}$ *whenever* $\mathbf{a} \preceq \mathbf{b}$.

So, one possible topologically sorted sequence would be: [x, z, y]. The other is [z, x, y].

Having now determined that the two strongly connected components are ordered as $\{\mathtt{x}, \mathtt{z}\} \preceq \{\mathtt{y}\}$, we may transform the original expression into the following one:

```
letrec
    y = 5
in letrec
       x = y + 7 + second z
       z = (x, y)
   in e
```

We next consider efficient ways to implement the strongly connected component algorithm.

6.8.2 Implementing a strongly connected component algorithm

Depth first search

We will first consider the problem of implementing a depth first search of a graph. The function `depthFirstSearch` is parameterised over the map from vertices to their offspring; this permits us to reverse the direction in which the edges are traversed. Furthermore, we choose to make the maps *ins* and *outs* into functions from vertices to sequences of vertices. The reason for this change is that we have to traverse the offspring in some order and this is easier to arrange if we have a sequence rather than a set.

```
> depthFirstSearch :: (* -> [*])   -> || Map
>                     (set *, [*]) -> || State: visited set,
>                                     ||        current sequence of vertices
>                     [*]          -> || Input vertices sequence
>                     (set *, [*])    || Final state
```

The function `depthFirstSearch` updates a state as it runs down the input vertex sequence, using `foldll`. The state consists of two parts: in the first we keep track of all of the vertices that we have so far visited; in the second we construct the sequence of vertices that we will output.

```
> depthFirstSearch
> = foldll . search
>   where
```

The key part of the depth first search is coded into `search`: if we have already visited the vertex then the state is unchanged.

```
>   search relation (visited, sequence) vertex
>   = (visited,          sequence ), setElementOf vertex visited
```

On the other hand, if this is the first time we have visited the vertex, then we must proceed to search from this vertex. When the state is returned from this search, we must add the current vertex to the sequence.

```
>   = (visited', vertex: sequence'), otherwise
>     where
```

The visited set must be updated with the current vertex, before we begin the search, and the list of vertices to search from is determined by applying the map `relation`.

```
>     (visited', sequence')
>     = depthFirstSearch relation
>                         (setUnion visited (setSingleton vertex), sequence)
>                         (relation vertex)
```

The result will be a set of vertices visited, and a sequence of these visited vertices, in topological sort order.

Exercise 6.14. Prove the following theorem:

$$\texttt{depthFirstSearch}\ \mathit{outs}\ (\{\}, [])\mathbf{S} = (\mathbf{V}, \mathbf{S}')$$

where $\mathbf{S}$ is a sequence of vertices, $\mathbf{V} = \bigcup_{\mathbf{v} \in \mathbf{S}} \mathit{outs}\ \mathbf{v}$, and $\mathbf{S}'$ is a topologically sorted sequence of the vertices in $\mathbf{V}$.

Spanning search

The function `spanningSearch` is a slight adaptation of the function `depthFirstSearch`, in which we retain the structuring information obtained during the search.

```
> spanningSearch   :: (* -> [*])        -> || The map
>                     (set *, [set *]) -> || Current state: visited set,
>                                         ||  current sequence of vertice sets
>                     [*]              -> || Input sequence of vertices
>                     (set *, [set *])    || Final state
```

Again, it is defined in terms of an auxiliary function `search`

```
> spanningSearch

> = foldll . search
>   where
```

If the current vertex has been visited already then we return the current state.

```
>   search relation (visited, setSequence) vertex
>   = (visited,          setSequence ), setElementOf vertex visited
```

Alternatively, if this is the first time we have visited the current vertex, then we search – using `depthFirstSearch` – from the current vertex. The sequence that is returned constitutes the component associated with the current vertex. We therefore add it to the sequence of sets that we are constructing.

```
>   = (visited', setFromList (vertex: sequence): setSequence), otherwise
>     where
>     (visited', sequence)
>     = depthFirstSearch relation
>                         (setUnion visited (setSingleton vertex), [])
>                         (relation vertex)
```

Strongly connected components

The strongly connected component algorithm can now be implemented as the following function:

```
> scc :: (* -> [*]) -> || The "ins"  map
>        (* -> [*]) -> || The "outs" map
>        [*]        -> || The root vertices
>        [set *]       || The topologically sorted components
```

The function `scc` consists of two passes over the graph.

```
> scc ins outs
> = spanning . depthFirst
```

In the first we construct a topologically sorted sequence of the vertices of the graph.

```
>   where depthFirst = second . depthFirstSearch outs (setEmpty, [])
```

In the second pass we construct the reverse of the topologically sorted sequence of strongly connected components.

```
>         spanning   = second . spanningSearch   ins  (setEmpty, [])
```

Let us consider what happens when we construct the first component. At the head of the sequence is the vertex `a`. Any other vertices that satisfy `a` $\preceq$ `b` will occur later in the sequence. There are no vertices satisfying `b` $\preceq$ `a`. The call to `spanningSearch` with `ins` as its relational parameter, will construct *ins** `a`. The visited set will be augmented with each vertex in the component.

In the example we considered earlier, we will be applying `spanning` to the list [`x, z, y`]. This expands to:

```
second (search ins (search ins (search ins ({},[]) x) z) y)
```

Expanding the inner `search` we obtain:

```
second (search ins (search ins (vs, [setFromList (x:s)]) z) y)
where (vs,s) = depthFirstSearch ins ({x},[]) (ins x)
```

But `ins x` = [`z`]; this means that `vs` = {`x, z`} and `s` = [`z`]. Hence we reduce the expression to:

```
second (search ins (search ins ({x,z}, [{x,z}]) z) y)
```

Because `z` is already in the visted set, this becomes:

```
second (search ins ({x,z}, [{x,z}]) y)
```

The `search` expands as:

```
second (vs, [setFromList (y:s), {x,z}])
where (vs,s) = depthFirstSearch ins ({x,y,z},[]) (ins y)
```

But `ins y` = [`x`, `z`]; both of which are already visited, so `vs` = {`x`, `z`, `y`} and `s` = [].

The final form of the expression becomes:

```
[{y}, {x,z}]
```

> *The visited set in* ***spanningSearch*** *represents those vertices that have already been assigned to a strongly connected component.*

When we come across a vertex in the input sequence that is already in the visited set, it behaves as if it had been deleted from further consideration. Suppose that the vertex `b` is the next vertex in the input sequence that has not already been visited. When we come to compute

$$\texttt{depthFirstSearch}\ \mathit{ins}\ (\{\texttt{b}\} \cup \texttt{visited}, [])\ (\mathit{ins}\ \texttt{b})$$

This will produce a new visited set (which will be *scc* `a` ∪ *scc* `b`) and a sequence whose elements are the vertices in *scc* `b`.

Note that the strongly connected components are output in reverse topological order.

Exercise 6.15. Let

$$\texttt{scc}\ \mathit{ins}\ \mathit{outs}\ \mathbf{R} = \mathbf{S},$$

and $\mathbf{V} = \bigcup\{\mathit{ins}^*\ \mathbf{v} \mid \mathbf{v} \in \mathbf{R}\}$. Prove that

- firstly, that the sequence **S** contains all of the strongly connected components, i.e.

$$\texttt{setFromList}\ \mathbf{S} = \bigcup\{\mathit{scc}\ \mathbf{v} \mid \mathbf{v} \in \mathbf{V}\};$$

- secondly, that these components are in reverse topological order, i.e. if $\texttt{a} \preceq \texttt{b}$ then `b` occurs before `a` in the sequence **S**.

6.8.3 A dependency analysis

We can now perform a dependency analysis on the program. Whenever we come across a `let(rec)` block we must split it into strongly connected components. In the case of `let`, this is simple; there are no dependencies so we simply separate the list of definitions into a singleton list for each definition.

The dependency analysis is performed by the function `dependency`. This uses information from a prior pass of `freeVars` to rearrange `let(rec)`s; this information is then used by the auxiliary function `depends`.

```
> dependency :: coreProgram -> coreProgram
> dependency =  depends . freeVars
```

```
> depends :: annProgram name (set name) -> coreProgram
> depends prog = [(name,args, depends_e rhs) | (name, args, rhs) <- prog]
```

The work is done by `depends_e`, whose only interesting case is that for `let(rec)`.

```
> depends_e :: annExpr name (set name) -> coreExpr
> depends_e (free, ANum n)          = ENum n
> depends_e (free, AConstr t a)     = EConstr t a
> depends_e (free, AVar v)          = EVar v
> depends_e (free, AAp e1 e2)       = EAp (depends_e e1) (depends_e e2)
> depends_e (free, ACase body alts) = ECase (depends_e body)
>                                             [ (tag, args, depends_e e)
>                                             | (tag, args, e) <- alts
>                                             ]
> depends_e (free, ALam ns body)    = ELam ns (depends_e body)
```

In the case of `letrec`s we must construct the dependency graph, and then apply the `scc` function to determine the way to split the definitions. If `defnGroups` is $[\mathbf{d}_1, \mathbf{d}_2, \ldots, \mathbf{d_n}]$ – and we are processing a `letrec` – then the `letrec` will be transformed to:

`letrec` $\mathbf{d}_1$ `in letrec` $\mathbf{d}_2$ `in` $\ldots$ `letrec` $\mathbf{d_n}$ `in body`.

```
> depends_e (free, ALet is_rec defns body)
> = foldr (mkDependLet is_rec) (depends_e body) defnGroups
>   where
>   binders    = bindersOf defns
```

The set of variables that we are interested in is derived from the `binders`, and is called the `binderSet`.

```
>   binderSet  = setFromList binders, is_rec
>              = setEmpty,            otherwise
```

From this we can construct the `edges` of the dependency graph.

```
>   edges      = [(n, f) | (n, (free, e)) <- defns;
>                          f <- setToList (setIntersection free binderSet)]
```

And thus the functions `ins` and `outs` required by the strongly connected component algorithm.

```
>   ins  v     = [u | (u,w) <- edges; v=w]
>   outs v     = [w | (u,w) <- edges; v=u]
```

The resulting list of sets is converted into a list of lists, called `components`.

```
>   components = map setToList (scc ins outs binders)
```

We construct the `defnGroups` by looking up the expression bound to each binder in the original definitions, `defns`:

```
>   defnGroups = [ [ (n, aLookup defns n (error "defnGroups"))
>                  | n <- ns]
>                | ns <- components
>                ]
```

Finally, to join together each group in `defnGroups`, we define `mkDependLet`, which recursively does dependency analysis on each right-hand side, and then builds the results into a `let(rec)` expression: A simple definition is:

```
> mkDependLet is_rec dfs e = ELet is_rec [(n, depends_e e) | (n,e) <- dfs] e
```

Exercise 6.16. In addition to the definition groups from non-recursive `lets` we sometimes get non-recursive definitions arising in `letrec`'s. This is the case for `y` in the example we have used.

Redefine `mkDependLet` to make such bindings with a `let` and not a `letrec`. Hint: use the free-variable information present in the right-hand sides of the definitions passed to `mkDependLet`.

6.9 Conclusion

It is interesting to compare our approach to full laziness with Bird's very nice paper [Bird 1987] which addresses a similar problem. Bird's objective is to give a formal development of an efficient fully lazy lambda lifter, by successive transformation of an initial specification. The resulting algorithm is rather complex, and would be hard to write down directly, thus fully justifying the effort of a formal development.

In contrast, we have expressed our algorithm as a composition of a number of very simple phases, each of which can readily be specified and written down directly. The resulting program has a constant-factor inefficiency, because it makes many traversals of the expression. This is easily removed by folding together successive passes into a single function, eliminating the intermediate data structure. Unlike Bird's transformations, this is a straightforward process.

Our approach has the major advantage that is is *modular*. For example:

- We were able to reuse existing functions on several occasions (`freeVars`, `rename`, `collectSCs` and so on).
- The multi-pass approach means that each pass has a well-defined, simple purpose, which makes it easier to modify. For example, we modified the `identifyMFEs` algorithm to be more selective about where full laziness is introduced (Section 6.7.3).

- We could use the major phases in various combinations to 'snap together' a variety of transformations. For example, we could choose whether or not to do dependency analysis and full laziness, and which lambda lifter to use, simply by composing the appropriate functions at the top level.

The main disadvantage of our approach is that we are unable to take advantage of one optimisation suggested by Hughes, namely ordering the parameters to a supercombinator to reduce the number of MFEs. The reason for this is that the optimisation absolutely requires that lambda lifting be entwined with the process of MFE identification, while we have carefully separated these activities! Happily for us, the larger MFEs created by this optimisation are always partial applications, which should probably *not* be identified as MFEs because no work is shared thereby (Section 6.7.3). Even so, matters might not have fallen out so fortuitously, and our separation of concerns has certainly made some sorts of transformation rather difficult.

Appendix A

Utilities module

This appendix gives definitions for various useful types and functions used throughout the book.

A.1 The heap type

The abstract data types `heap` and `addr` are used to represent the garbage-collected heap of nodes for each of our implementations.

A.1.1 Specification

A `heap` of `*` is a collection of *objects* of type `*`, each identified by a unique *address* of type `addr`. The following operations are provided:

```
> abstype heap *, addr
> with
>       hInitial :: heap *
>       hAlloc   :: heap * -> * -> (heap *, addr)
>       hUpdate  :: heap * -> addr -> * -> heap *
>       hFree    :: heap * -> addr -> heap *
```

`hInitial` returns an initialised empty heap. `hAlloc` takes a heap and an object, and returns a new heap and an address; the new heap is exactly the same as the old one, except that the specified object is found at the address returned. `hUpdate` takes a heap, an address and an object; it returns a new heap in which the address is now associated with the object. `hFree` takes a heap and an address and returns a new heap with the specified object removed.

```
>       hLookup  :: heap * -> addr -> *
>       hAddresses :: heap * -> [addr]
>       hSize    :: heap * -> num
```

`hLookup` takes a heap and an address and returns the object associated with that address. `hAddresses` returns the addresses of all the objects in the heap. `hSize` returns the number of objects in the heap.

```
>       hNull       :: addr
>       hIsnull     :: addr -> bool
```

hNull is an address guaranteed to differ from every address returned by hAlloc; hIsnull tells whether an address is this distinguished value.

Finally, we add a show function so that addresses can be printed easily.

```
>        showaddr :: addr -> [char]
```

By giving it the name show followed by the name of the type (addr), we inform Miranda that when Miranda's built-in show function encounters an object of type addr, it should use showaddr to convert it to a list of characters.

A.1.2 Representation

The heap is represented as a triple, containing:

- the number of objects in the heap;
- a list of unused addresses;
- an association list mapping addresses to objects.

Addresses are represented as numbers.

```
> heap * == (num, [num], [(num, *)])
> addr   == num
```

We implement the operations in a (fairly) obvious manner.

```
> hInitial                             = (0,      [1..],  [])
> hAlloc  (size, (next:free), cts) n   = ((size+1, free,   (next,n) : cts),next)
> hUpdate (size, free,        cts) a n = (size,   free,   (a,n) : remove cts a)
> hFree   (size, free,        cts) a   = (size-1, a:free, remove cts a)


> hLookup (size,free,cts) a
>  = aLookup cts a (error ("can't find node " ++ showaddr a ++ " in heap"))
>
> hAddresses (size, free, cts) = [addr | (addr, node) <- cts]
>
> hSize (size, free, cts) = size


> hNull    = 0
> hIsnull a = a=0
> showaddr a = "#" ++ shownum a         || Print # to identify addresses
```

The auxiliary function `remove` removes an item from a heap contents:

```
> remove :: [(num,*)] -> num -> [(num,*)]
> remove [] a = error ("Attempt to update or free nonexistent address #" ++
>                      shownum a)
> remove ((a',n):cts) a = cts,                          a=a'
>                       = (a',n) : remove cts a,        otherwise
```

A.2 The association list type

An *association list* associates *keys* to *values*. It is represented by a list of (key,value) pairs, using a type synonym. It is not an abstract type because it turns out to be so convenient to use list-manipulation operations on it.

```
> assoc * ** == [(*,**)]
```

You can use one association list, $\mathbf{e}_1$, to extend another, $\mathbf{e}_2$, using ordinary list append, thus $\mathbf{e}_1$ `++` $\mathbf{e}_2$. A lookup in this extended environment will search $\mathbf{e}_1$ first and then $\mathbf{e}_2$.

Given a key, **k**, you can find the associated value using `aLookup`.

```
> aLookup :: assoc * ** -> * -> ** -> **
```

The call `aLookup` **alist key default** searches the association list **alist** starting from the head of the list; if it finds a (**key**, **val**) pair it returns **val**, otherwise it returns **default**.

```
> aLookup []         k' def = def
> aLookup ((k,v):bs) k' def = v,                        k = k'
>                           = aLookup bs k' def,        otherwise
```

The functions `aDomain` and `aRange` find the range and domain of the association list, respectively:

```
> aDomain :: assoc * ** -> [*]
> aDomain alist = [key | (key,val) <- alist]
>
> aRange :: assoc * ** -> [**]
> aRange  alist = [val | (key,val) <- alist]
```

`aEmpty` is the empty association list:

```
> aEmpty = []
```

A.3 Generating unique names

In Chapter 6 we need to generate unique names for newly generated supercombinators. The abstract data type `nameSupply` acts as a supply of unique names.

```
> abstype nameSupply with
>       getName  :: nameSupply -> [char]   -> (nameSupply, [char])
>       getNames :: nameSupply -> [[char]] -> (nameSupply, [[char]])
>       initialNameSupply :: nameSupply
```

There are three operations. `getName` takes a name supply and a prefix string, and returns a depleted name supply together with a string which is a new unique name; this string has the specified prefix. `getNames` does the same thing for a list of prefixes. Finally, `initialNameSupply` is the initial, undepleted name supply.

A.3.1 Representation

A name supply is represented by a single integer.

```
> nameSupply == num
> initialNameSupply = 0
> getName name_supply prefix = (name_supply+1, makeName prefix name_supply)
> getNames name_supply prefixes
>  = (name_supply + #prefixes, zipWith makeName prefixes [name_supply..])


> makeName prefix ns = prefix ++ "_" ++ shownum ns
```

A.4 Sets

The abstract data type of sets has the following signature.

```
> abstype set *
>  with setFromList     :: [*]     -> set *
>       setToList       :: set *   -> [*]
>       setUnion        :: set *   -> set * -> set *
>       setIntersection :: set *   -> set * -> set *
>       setSubtraction  :: set *   -> set * -> set *
>       setElementOf    :: *       -> set * -> bool
>       setEmpty        :: set *
>       setIsEmpty      :: set *   -> bool
>       setSingleton    :: *       -> set *
>       setUnionList    :: [set *] -> set *
```

A.4.1 Representation

In this implementation, sets are represented by *ordered* lists.

```
> set * == [*]           || Ordered by the sort function
```

The implementation of the operations is straightforward.

```
> setEmpty = []
> setIsEmpty s = s = []
> setSingleton x = [x]


> setFromList = rmdup . sort
>                 where rmdup []       = []
>                       rmdup [x]      = [x]
>                       rmdup (x:y:xs) = rmdup (y:xs),    x=y
>                                      = x: rmdup (y:xs), otherwise


> setToList xs = xs


> setUnion []     []           = []
> setUnion []     (b:bs)       = (b:bs)
> setUnion (a:as) []           = (a:as)
> setUnion (a:as) (b:bs)       = a: setUnion as (b:bs), a < b
>                               = a: setUnion as bs,     a = b
>                               = b: setUnion (a:as) bs, a > b


> setIntersection []     []     = []
> setIntersection []     (b:bs) = []
> setIntersection (a:as) []     = []
> setIntersection (a:as) (b:bs) = setIntersection as (b:bs), a < b
>                                = a: setIntersection as bs,  a = b
>                                = setIntersection (a:as) bs, a > b


> setSubtraction []     []      = []
> setSubtraction []     (b:bs)  = []
> setSubtraction (a:as) []      = (a:as)
> setSubtraction (a:as) (b:bs)  = a: setSubtraction as (b:bs), a < b
>                                = setSubtraction as bs,        a = b
>                                = setSubtraction (a:as) bs,    a > b


> setElementOf x []            = False
> setElementOf x (y:ys)        = x=y \/ (x>y & setElementOf x ys)


> setUnionList = foldll setUnion setEmpty
```

A.5 Other useful function definitions

The definitions of fst and snd are present in later versions of Miranda, but not earlier ones. We always use first and second instead to avoid compatibility problems.

```
> first (a,b) = a
> second (a,b) = b
```

The function zipWith zips together two lists, combining corresponding elements with a given function. The resulting list is as long as the shorter of the two input lists.

```
> zipWith :: (* -> ** -> ***) -> [*] -> [**] -> [***]
> zipWith f []     ys     = []
> zipWith f xs     []     = []
> zipWith f (x:xs) (y:ys) = f x y: zipWith f xs ys
```

The definition of foldl differs between different versions of Miranda, so we avoid the problem by writing our own function foldll, which does the following: given a dyadic function $\otimes$, a value $\mathbf{b}$ and a list $\mathbf{xs} = [\mathbf{x}_1, ..., \mathbf{x_n}]$, foldll $\otimes$ $\mathbf{b}$ $\mathbf{xs}$ computes $(\ldots((\mathbf{b} \otimes \mathbf{x}_1) \otimes \mathbf{x}_2) \otimes \ \ldots \mathbf{x_n})$. Section 1.5.1 contains a simple example of foldll in action, together with a picture.

```
> foldll :: (* -> ** -> *) -> * -> [**] -> *
> foldll f b []     = b
> foldll f b (x:xs) = foldll f (f b x) xs
```

Finally, the function mapAccuml is a rather useful combination of map and foldll. It is given a function, an accumulator and a list. For each element of the list it applies the function to the current accumulator and that list element, which gives a new value of the accumulator and a new list element. The result of mapAccuml is the final value of the accumulator, and the list of all the results. The 'l' in the function name says that the accumulator is passed along from left to right. Section 2.3.4 has an example of mapAccuml in action, together with a picture.

```
> mapAccuml :: (* -> ** -> (*, ***)) || Function of accumulator and element
>                                    ||   input list, returning new
>                                    ||   accumulator and element of result list
>              -> *                  || Initial accumulator
>              -> [**]               || Input list
>              -> (*, [***])         || Final accumulator and result list
>
> mapAccuml f acc []     = (acc, [])
> mapAccuml f acc (x:xs) = (acc2, x':xs')
>                          where (acc1, x')  = f acc x
>                                (acc2, xs') = mapAccuml f acc1 xs
```

Appendix B

Example Core-language programs

In this Appendix we give a few Core-language programs which are useful for testing some of the implementations developed in the book. They assume that the functions defined in the prelude (Section 1.4) are defined.

B.1 Basic programs

The programs in this section require only integer constants and function application.

B.1.1 Ultra-basic tests

This program should return the value `3` rather quickly!

```
main = I 3
```

The next program requires a couple more steps before returning `3`.

```
id = S K K ;
main = id 3
```

This one makes quite a few applications of `id` (how many?).

```
id = S K K ;
main = twice twice twice id 3
```

B.1.2 Testing updating

This program should show up the difference between a system which does updating and one which does not. If updating occurs, the evaluation of `(I I I)` should take place only once; without updating it will take place twice.

```
main = twice (I I I) 3
```

B.1.3 A more interesting example

This example uses a functional representation of lists (see Section 2.8.3) to build an infinite list of 4's, and then takes its second element. The functions for head and tail (`hd` and `tl`) return `abort` if their argument is an empty list. The `abort` supercombinator just generates an infinite loop.

```
cons a b cc cn = cc a b ;
nil      cc cn = cn ;
hd list = list K abort ;
tl list = list K1 abort ;
abort = abort ;

infinite x = cons x (infinite x) ;

main = hd (tl (infinite 4))
```

B.2 `let` and `letrec`

If updating is implemented, then this program will execute in fewer steps than if not, because the evaluation of `id1` is shared.

```
main = let id1 = I I I
       in id1 id1 3
```

We should test nested `let` expressions too:

```
oct g x = let h = twice g
          in let k = twice h
          in k (k x) ;
main = oct I 4
```

The next program tests `letrec`s, using 'functional lists' based on the earlier definitions of `cons`, `nil`, etc.

```
infinite x = letrec xs = cons x xs
             in xs ;
main = hd (tl (tl (infinite 4)))
```

B.3 Arithmetic

B.3.1 No conditionals

We begin with simple tests which do not require the conditional.

```
main = 4*5+(2-5)
```

This next program needs function calls to work properly. Try replacing `twice twice` with `twice twice twice` or `twice twice twice twice`. Predict what the result should be.

```
inc x = x+1;
main = twice twice inc 4
```

Using functional lists again, we can write a length function:

```
length xs = xs length1 0 ;
length1 x xs = 1 + (length xs) ;

main = length (cons 3 (cons 3 (cons 3 nil)))
```

B.3.2 With conditionals

Once we have conditionals we can at last write 'interesting' programs. For example, factorial:

```
fac n = if (n==0) 1 (n * fac (n-1)) ;
main = fac 5
```

The next program computes the greatest common divisor of two integers, using Euclid's algorithm:

```
gcd a b = if (a==b)
                a
          if (a<b) (gcd b a) (gcd b (a-b)) ;
main = gcd 6 10
```

The `nfib` function is interesting because its result (an integer) gives a count of how many function calls were made during its execution. So the result divided by the execution time gives a performance measure in function calls per second. As a result, `nfib` is quite widely used as a benchmark. The 'nfib-number' for a particular implementation needs to be taken with an enormous dose of salt, however, because it is critically dependent on various rather specialised optimisations.

```
nfib n = if (n==0) 1 (1 + nfib (n-1) + nfib (n-2)) ;
main = nfib 4
```

B.4 Data structures

This program returns a list of descending integers. The evaluator should be expecting a list as the result of the program. `cons` and `nil` are now expected to be implemented in the prelude as `Pack{2,2}` and `Pack{1,0}` respectively.

```
downfrom n = if (n == 0)
                  nil
                  (cons n (downfrom (n-1))) ;
main = downfrom 4
```

The next program implements the Sieve of Eratosthenes to generate the infinite list of primes, and takes the first few elements of the result list. If you arrange that output is printed incrementally, as it is generated, you can remove the call to `take` and just print the infinite list.

```
main = take 3 (sieve (from 2)) ;

from n = cons n (from (n+1)) ;

sieve xs = case xs of
                <1> -> nil ;
                <2> p ps -> cons p (sieve (filter (nonMultiple p) ps)) ;

filter predicate xs
        = case xs of
                <1> -> nil ;
                <2> p ps -> let rest = filter predicate ps
                            in
                            if (predicate p) (cons p rest) rest ;

nonMultiple p n = ((n/p)*p) ~= n ;

take n xs = if (n==0)
                nil
                (case xs of
                        <1> -> nil ;
                        <2> p ps -> cons p (take (n-1) ps))
```

Bibliography

[Aho *et al.* 1986] Aho, A.V., R. Sethi and J. D. Ullman, (1986) *Compilers: principles, techniques and tools*, Addison Wesley.

[Argo 1989] Argo, G. (1989) *Improving the three instruction machine*, Functional Programming Languages and Computer Architecture, Addison Wesley.

[Argo 1991] Argo, G. (1991) *Efficient laziness*, Ph.D. thesis, Department of Computing Science, University of Glasgow, Dec. 1991 (to appear).

[Augustsson 1984] Augustsson, L. (1984) A Compiler for Lazy ML, in *Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming*, Austin, Texas, Aug. 1984, pp. 218–227.

[Augustsson 1987] Augustsson, L. (1987) *Compiling lazy functional languages, part II*, Ph.D. thesis, Chalmers Tekniska Högskola, Göteborg, Sweden.

[Baker 1978] Baker, H. (1978) List processing in real time on a serial computer, *Communications of the ACM* **21**(4), 280–294.

[Bird 1987] Bird, R. (1987) A formal development of an efficient supercombinator compiler, *Science of Computer Programming* **8**, 113–137.

[Bird and Wadler 1988] Bird, R., and P. L. Wadler, (1988) *Introduction to functional programming*, Prentice Hall.

[Burn 1991] Burn, G.L. (1991) *Lazy functional languages: abstract interpretation and compilation*, Pitman.

[Cheney 1970] Cheney, C. J. (1970) A non-recursive list compaction algorithm, *Communications of the ACM* **13**(11), 677–678.

[Cohen 1981] Cohen, J. (1981) Garbage collection of linked data structures, *ACM Computing Surveys* **13**(3), 341–367.

[Fairbairn 1985] Fairbairn, J. (1985) Removing redundant laziness from supercombinators, *Proceedings of the Aspenas workshop on implementation of functional languages*, Chalmers University, Feb. 1985.

[Fairbairn 1986] Fairbairn, J. (1986) *Making form follow function – an exercise in functional programming style*, TR 89, Computer Lab., Cambridge.

[Fairbairn and Wray 1987] Fairbairn, J. and S. Wray, (1987) *TIM – a simple lazy abstract machine to execute supercombinators*, Functional Programming Languages and Computer Architecture, LNCS 274, Springer Verlag.

[Fenichel and Yochelson 1969] Fenichel, R. R. and J. C. Yochelson, (1969) A Lisp garbage collector for virtual memory computer systems, *Communications of the ACM* **12**(11), 611–612.

[Goldberg 1988] Goldberg, B. F. (1988) *Multiprocessor execution of functional programs*, Ph.D. thesis, YALEU/DCS/RR-618, Department of Computer Science, Yale University, April 1988.

[Holst 1990] Holst, C. K. (1990) Improving full laziness, in *Functional Programming, Glasgow 1990*, ed. Peyton Jones, Hutton & Holst, Workshops in Computing, Springer Verlag.

[Holyer 1991] Holyer, I. (1991) *Functional programming with Miranda*, Pitman.

[Hughes 1983] Hughes, R. J. M. (1983) *The design and implementation of programming languages*, D.Phil. thesis, Programming Research Group, Oxford, July 1983.

[Johnsson 1984] Johnsson, T. (1984) Efficient compilation of lazy evaluation, in *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*, Montreal, Canada, June 1984, pp. 58–69.

[Johnsson 1985] Johnsson, T. (1985) Lambda lifting: transforming programs to recursive equations, in *Proceedings of the IFIP Conference on Functional Programming and Computer Architecture*, ed. Jouannaud, LNCS 201, Springer Verlag, pp. 190–205.

[Johnsson 1987] Johnsson, T. (1987) *Compiling lazy functional languages*, Ph.D. thesis, Chalmers Tekniska Högskola, Göteborg, Sweden.

[Jones *et al.* 1989] Jones, N. D., P. Sestoft and H. Søndergaard, (1989) Mix: a self-applicable partial evaluator for experiments in compiler generation, *Lisp and Symbolic Computation* **2**(1), 9–50.

[Kingdon *et al* 1991] Kingdon, H., D. Lester and G. L. Burn, (1991) The HDG-machine: a highly distributed graph reducer for a transputer network, *The Computer Journal* **34**(4), 290–302.

[Lester 1988] Lester, D. R. (1988) *Combinator graph reduction: a congruence and its applications*, D.Phil. Thesis, Technical Monograph PRG-73, Programming Research Group, Keble Rd, Oxford.

[Peyton Jones 1987] Peyton Jones, S. L. (1987) *The Implementation of Functional Programming Languages*. Prentice Hall International Series in Computer Science. Prentice Hall, Hemel Hempstead.

[Peyton Jones 1989] Peyton Jones, S. L. (1989) Parallel implementations of functional programming languages, *The Computer Journal* **32**(2), 175–186.

[Peyton Jones 1991] Peyton Jones, S. L. (1991) The Spineless Tagless G-machine: a second attempt, in *Proceedings of the Workshop on Parallel Implementation of Functional Languages*, ed. Glaser & Hartel, CSTR 91–07, Department of Electronics and Computer Science, University of Southampton.

[Peyton Jones and Lester 1991] Peyton Jones, S. L. and D. Lester, (1991) A modular, fully lazy lambda lifter in Haskell, *Software – Practice and Experience* **21**(5), 479–506.

[Peyton Jones and Salkild 1989] Peyton Jones, S. L. and J. Salkild, (1989) The Spineless Tagless G-machine, in *Functional Programming Languages and Computer Architecture*, ed. MacQueen, Addison Wesley.

[Schorr and Waite 1967] Schorr, H. and W. Waite, (1967) An efficient machine-independent procedure for garbage collection, *Communications of the ACM* **10**(8), 501–506.

[Wadler 1985] Wadler, P. L. (1985) How to replace failure by a list of successes, in *Functional Programming Languages and Computer Architecture*, Nancy, LNCS 201, Springer Verlag, pp. 113–128.

[Wadler 1987] Wadler, P. L. (1987) Projections for Strictness Analysis, in *Proceedings of the Functional Programming Languages and Computer Architecture Conference*, Portland, LNCS 274, Springer Verlag, pp. 385–407.

[Wakeling and Dix 1989] Wakeling, D. and A. Dix (1989) *Optimising partial applications in TIM*, Department of Computer Science, University of York, March 1989.

Subject index

Underlined entries in the index indicate where terms or compilation schemes are defined.

Code index

This index indicates the point of definition of every Miranda function defined in the book (underlined entries), and every reference to typewriter-font material in running text (non-underlined entries).