# Compiling Functional Programs to Java Byte-code

Gary Meehan
Department of Computer Science
University of Warwick
E-mail: garym@dcs.warwick.ac.uk

September 2, 1997

**Abstract**

The aim of the G-machine is to deconstruct a functional program, represented as a graph, into a list of linear instructions — G-Code — which, when executed, will construct an equivalent graph and reduce it into Weak Head Normal Form. The Java Virtual Machine (JVM) provides a machine-independent execution environment which executes Java *byte-code*. This byte-code is essentially a machine code for object-oriented programs. It was designed as the target of a Java compiler, but there is no reason why it cannot be used as the target of other languages. In this report we shall look at compiling functional programs down to the JVM, using the G-machine as a guide.

## 1 Introduction

The aim of the G-machine [8] is to deconstruct a functional program, represented as a graph, into a list of linear instructions — G-Code — which, when executed, will construct an equivalent graph and reduce it into Weak Head Normal Form [3]. The source code of the G-Machine is a set of supercombinators [8], with one particular zero-argument supercombinator, i. e. a Constant Applicative Form (CAF), denoted as `main` in Ginger, used as the starting point for the program. Each of these supercombinators is compiled into its own separate list of G-Code instructions.

The G-Machine consists of 4 different components:

**S** The Stack. Used to store working results.

**G** The Graph. A globally accessible space containing all the nodes of the graph being worked on.

**C** The Code. The instructions to execute.

**D** The Dump. Used as a store while working on different parts of the graph.

When execution has terminated, the result of reducing the graph, i. e. evaluating `main`, will reside on top of the stack.

The Java Virtual Machine (JVM) [7] provides a machine-independent execution environment which executes Java *byte-code*. This byte-code is essentially a machine code for object-oriented programs. It was designed as the target of Java compilers, but there is no reason why it cannot be used as the target of other languages.

In this report we shall look at compiling functional programs down to the JVM, using the G-machine as a guide. In effect, we will have a G-machine running on the JVM. Our source will be the functional language Ginger [5]. Our eventual aim is to compile the reduction rules for our supercombinators down to a class of Java static methods, one class per compiled Ginger program. The `main` method of this class will correspond to the `main` supercombinator of our source program.

The order of the concepts dealt with in this report follows roughly the order in which the compiler proceeds. Section 2 covers the representation of the components of the G-Machine in the JVM. Section 3 deals with parsing our source language and the transformations — dependency analysis, lambda-lifting, etc. — needed to get our code into a form which is easily compilable. Section 4 deals with the actual compilation of our transformed program and details the compilation schemes used. Section 5 deals with the implementation of the *operations* of the G-machine, namely evaluating and unwinding a graph, printing results and implementing primitive functions. Sections 6 presents some sample results and 7 concludes.

# 2 Representing the G-Machine in the Java Virtual Machine

The JVM provides an object-oriented view of programming. All code is stored in methods of some class. These methods come in two varieties: class (static) methods and instance (virtual) methods. The latter are messages rather than functions: they are passed to some object instance (along with any extra parameters) and return a (possibly void) result.

Each method comes with its own stack, used to hold values for JVM instructions and method calls, and a set of local variables, used to hold the method's parameters and any working results.

## 2.1 The Stack

Although it would seem natural to expect a 1-1 mapping between the G-Machine stack and the JVM stack, this is unfortunately not the case. The JVM stack has many restrictions placed upon it (for verification reasons) and some of the common operations (size of stack, empty stack, etc.) used by the G-Machine stack are not available.

However, the the G-Machine stack has two distinct usages: to hold values while constructing the graph and to store the ribs of a graph whilst we a performing a graph walk (see Section 5.2). The operations needed to construct the graph can be achieved using the JVM stack. However, the operations needed to reduce the graph are more complicated, and we also require that the stack be passed between method calls, something that cannot be done using the JVM stack. Therefore we split the G-Machine stack into two stacks: the standard JVM stack for construction, and instances of the core Java class `java.util.Stack` [10] for unwinding the graph. We shall refer to the latter as the R-stack.

## 2.2 The Graph

The G-Machine represents each node in the graph as a pointer to the actual data. The use of the pointer is important since a large part of the speed of the G-Machine is the ability to rewrite sections of the graph by re-assigning pointers so as to guarantee fully-lazy evaluation (i. e. everything is evaluated *at most* once). This suggests the natural use of pointers to objects in an object-oriented model. However, Java does not provide for the manipulation of pointers (*references* in the Java vernacular), their only use is to refer to objects which can *only* be referred to *via* pointers. We thus need to box the nodes of the graph in another class, the `Node` class, which will provide the functionality of the node pointers in the G-Machine.

We provide a class `Node` which will contain the nodes of the graph. These nodes will be primitives (numbers, characters, etc.), functions, application nodes, list constructor (cons) nodes and the empty list. The basic skeleton of the `Node` class can be seen in Appendix B.

### 2.2.1 Primitive nodes

The primitive Ginger types — integers, floating point numbers, booleans, characters and strings — are represented using the corresponding Java objects `Strings` — `Long`, `Double`, `Boolean` and `Character`. Note that we use Java double-width (64-bit) integers and floating point numbers to represent integers and floats in Ginger. This extra precision allows for the development of more interesting functional programs.

### 2.2.2 Function nodes

Functions are represented using the `Method` class in the `java.lang.reflect` package [10]. These `Method` instances correspond to the static methods of the class files created from compiling Ginger programs. The `Method` class is a way of incorporating functions as first class objects into Java, allowing us to examine the parameter and return types of a method, determine if the method has any modifiers (e. g. `public`, `static`, etc.), determine the defining class of the method, and to invoke the method. The `Method` class is generally used in co-operation with the `Class` class [10] which, amongst other things, has methods for determining the methods defined by that class.

### 2.2.3 Application, Cons and Empty List nodes

Applications, cons nodes and the empty list node need simple classes created for them as no applicable classes are available as standard in the Java API. For example, we define the `App` class *viz*:

```
public class App {
  public Node functor;
  public Node argument;

  public App(Node a, Node f) {
    argument = a;
    functor = f;
  }
}
```

The `Cons` class, representing cons nodes is similar. Note the ordering of the arguments in the constructor function, suggested by [8] (instances of `Cons` similarly take the tail of the list first). Although we could use the `null` class to serve such a purpose, we find it useful to be able to distinguish between the empty list and a `null` object (the latter only arising as the result of an error) which motivates the use of an `EmptyList` class.

## 2.3 The Code

The G-code, used to construct the graph, is replaced by JVM (or, rather, Jasmin [7]) instructions, which construct new objects using the `Node` classes and its subclasses (see above). These instructions will be the eventual target of our compiler.

The evaluating and printing of nodes is handled by the `eval` and `print` methods in the `Node` class. These are discussed in Sections 5.1 and 5.3.

## 2.4 The Dump

There is no need for an explicit dump in our compiler as its operations (storing the return addresses and stack pointers of functions) are handled internally by the JVM.

# 3 Parsing, Dependency Analysis and Lambda-Lifting

*Parsing* (Section 3.1) a program involves reading the source text and building the graph it represents in memory. While parsing we choose to do so elementary simplifications to the code to make it more amenable to compilation. *Dependency Analysis* (Section 3.2) involves simplifying blocks of local definitions so that they are easier to compile and produce more efficient code. *Lambda-lifting* (Section 3.3) is the process by which lambda nodes are removed from the graph and replaced with extra supercombinator definitions, yielding a program which is defined only by supercombinators.

## 3.1 Parsing

The source language of our compiler is Ginger [5]. This is a simple language with no pattern-matching or list comprehensions. There are facilities for parallelism and divide-and-conquer lists [2], but we do not implement these. The BNF of Ginger is detailed in Appendix A.

Each supercombinator definition is parsed into a `ParsedSuperComb` object using the $\mathcal{PS}$ scheme (see Section 3.1.1). The definition body of each of these is then parsed into instances of the subclasses of `ParsedExpr` using the $\mathcal{PE}$ scheme (see Sections 3.1.2–3.1.8). The class hierarchy of the target parsed objects can be see in Figure 1. Those nodes marked * are introduced at the dependency-analysis stage and not during the original parse. Those nodes marked ** will be eliminated during lambda-lifting and do not thus need to be compiled.

### 3.1.1 Parsing Supercombinators

We absorb any leading lambdas as extra parameters of the supercombinators. We have:

$$\mathcal{PS}[f\ x_1\ \ldots\ x_n\ \texttt{=}\ \backslash y_1\ \ldots\backslash\ y_m\ e] = \texttt{new ParsedSuperComb(}f\texttt{,}\ < x_1,\ldots,x_n,y_1,\ldots,y_m >\texttt{,}\ \mathcal{PE}[e]\texttt{)}$$

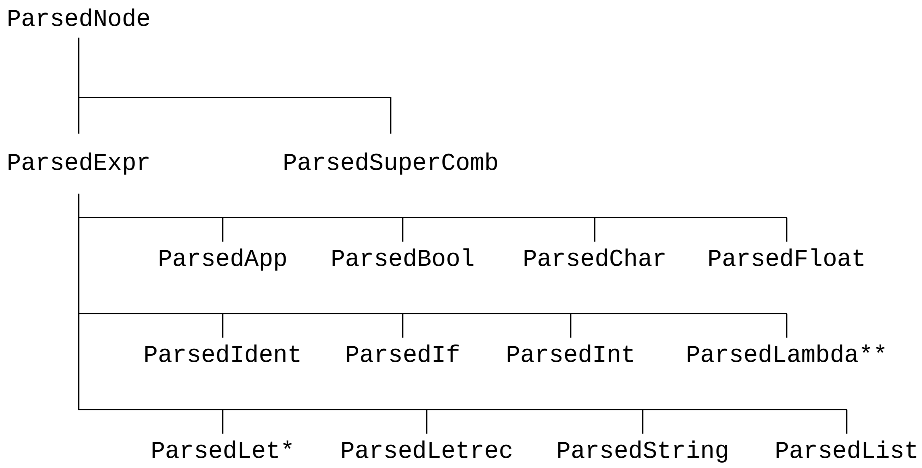


Figure 1: The parsed classes hierarchy

The constructor function of `ParsedSuperComb` takes as arguments the name of the supercombinator, its arguments (as an array, indicated by the $< \ldots >$) and its definition, and creates an instance of the appropriate object.

### 3.1.2 Parsing Primitives

Parsing primitives requires just the creation of the appropriate object. We have:

$$\begin{array}{rcll}
\mathcal{PE}[i] & = & \texttt{new ParsedInt(}i\texttt{)} & \text{, integer } i \\
\mathcal{PE}[x] & = & \texttt{new ParsedFloat(}x\texttt{)} & \text{, float } x \\
\mathcal{PE}[c] & = & \texttt{new ParsedChar(}i\texttt{)} & \text{, character } c \\
\mathcal{PE}[b] & = & \texttt{new ParsedBool(}b\texttt{)} & \text{, boolean } b \\
\mathcal{PE}[\texttt{"}s\texttt{"}] & = & \texttt{new ParsedString(}s\texttt{)} & \text{, string } s
\end{array}$$

### 3.1.3 Parsing Identifiers

An identifier is either a function or a variable name. Variables must be alphanumeric and must start with either a lowercase letter or an underscore [1]. User-defined functions also follow this pattern. However primitive functions can also be symbolic, e.g. `*`, `++`. Since these functions will eventually be represented as Java methods, and such symbols aren't allowed in Java method names, these symbolic names must be converted into alphanumeric form (or *romanized*). We do this in the obvious way, but choose to prefix the results of all such conversions with an underscore. This allows us to differentiate between the function `_and`, which is logical conjunction, and the function `and` which is the function which folds the conjunction over a list. The full list of conversions can be seen in Table 1. The parse scheme for identifiers is thus:

$$\mathcal{PE}[id] = \texttt{new ParsedIdent(}R(i)\texttt{)}$$

where $R$ romanizes the operator symbols (and leaves other functions alone).

### 3.1.4 Parsing Applications

Infix operators are converted into prefix romanized form (see above). A cons operator '`:`' with both of its arguments is parsed directly into a list.

$$\begin{array}{rcl}
\mathcal{PE}[f\ a] & = & \texttt{new ParsedApp(}\mathcal{PE}[f]\texttt{, }\mathcal{PE}[a]\texttt{)} \\
\mathcal{PE}[h : t] & = & \texttt{new ParsedList(}\mathcal{PE}[h]\texttt{, }\mathcal{PE}[t]\texttt{)}
\end{array}$$

Note, that unlike the `App` and `Cons` node, the constructors of `ParsedApp` and `ParsedList` take the functor/head argument first.

[1]Identifiers commencing with an uppercase letter are reserved for use as types.

| Symbol | Romanized Form | Symbol | Romanized Form |
|---|---|---|---|
| `:` | `_cons` | `~=` | `_ne` |
| `++` | `_cat` | `+` | `_plus` |
| `\|` | `_or` | `-` | `_minus` |
| `&` | `_and` | `%` | `_mod` |
| `<` | `_lt` | `*` | `_times` |
| `<=` | `_le` | `/` | `_divide` |
| `==` | `_eq` | `^` | `_pow` |
| `>` | `_gt` | `.` | `compose` |
| `>=` | `_ge` | `!` | `_list_index` |
| `~` | `_not` | `#` | `_length` |

Table 1: Romanized form of the symbolic operators

### 3.1.5 Parsing Lists

We have the standard comma-separated list constructions plus the four variants of the 'dot-dot' notation. The former are converted into multiple applications of the cons function (':') dealt with in Section 3.1.4; the latter into appropriate function calls, these functions being defined in the Prelude file (see Section 5.4).

$$
\begin{aligned}
\mathcal{PE}[\texttt{[]}] &= \texttt{new ParsedList()} \\
\mathcal{PE}[\texttt{[}e_1\texttt{,}e_2\texttt{,}\ldots\texttt{,}e_n\texttt{]}] &= \mathcal{PE}[e_1 : e_2 : \ldots : e_n : \texttt{[]}] \\
\mathit{Parse}[\texttt{[}e_1\texttt{..]}] &= \texttt{new ParsedApp(\_from, } \mathcal{PE}[e_1]\texttt{)} \\
\mathcal{PE}[\texttt{[}e_1\texttt{,}e_2\texttt{..]}] &= \texttt{new ParsedApp(\_fromThen, } \mathcal{PE}[e_1]\texttt{, } \mathcal{PE}[e_2]\texttt{)} \\
\mathcal{PE}[\texttt{[}e_1\texttt{..}e_2\texttt{]}] &= \texttt{new ParsedApp(\_fromTo, } \mathcal{PE}[e_1]\texttt{, } \mathcal{PE}[e_2]\texttt{)} \\
\mathcal{PE}[\texttt{[}e_1\texttt{,}e_2\texttt{..}e_3\texttt{]}] &= \texttt{new ParsedApp(\_fromThenTo, } \mathcal{PE}[e_1]\texttt{, } \mathcal{PE}[e_2]\texttt{, } \mathcal{PE}[e_3]\texttt{)}
\end{aligned}
$$

The zero-argument construct function of `ParsedList` constructs an empty list. The various constructor functions of `ParsedApp` construct multiple-argument applications in the expected manner.

### 3.1.6 Parsing Lambdas

We simply create a new `ParsedLambda` node, *viz*:

$$\mathcal{PE}[\backslash x\ e] = \texttt{new ParsedLambda(}\mathcal{PE}[x]\texttt{, } \mathcal{PE}[e]\texttt{)}$$

### 3.1.7 Parsing Conditionals

We convert any `elsif` blocks in the conditional into nested `if` statements. We have:

$$
\mathcal{PE}\left[\begin{array}{l}
\texttt{if } a_1 \texttt{ then } t_1 \\
\texttt{elsif } a_2 \texttt{ then } t_2 \\
\ldots \\
\texttt{elsif } a_{n-1} \texttt{ then } t_{n-1} \\
\texttt{else } d \\
\texttt{endif}
\end{array}\right]
=
\begin{array}{l}
\texttt{new ParsedIf(}\mathcal{PE}[a_1]\texttt{,}\mathcal{PE}[t_1]\texttt{,} \\
\quad \texttt{(new ParsedIf(}\mathcal{PE}[a_2]\texttt{,}\mathcal{PE}[t_2]\texttt{,} \\
\quad\quad \ldots \\
\quad\quad\quad \texttt{(new ParsedIf(}\mathcal{PE}[a_{n-1}]\texttt{,}\mathcal{PE}[t_{n-1}]\texttt{,}\mathcal{PE}[d]\texttt{))}\ldots\texttt{))}
\end{array}
$$

### 3.1.8 Parsing Local Definitions

All `let`s are presumed to be mutually recursive and are parsed as '`letrec`'s [8]; `where` blocks are also turned into `letrec`s *viz*:

$$\mathcal{PE}[e \texttt{ where } D \texttt{ endwhere}] = \mathcal{PE}[\texttt{let } D \texttt{ in } e \texttt{ endlet}]$$

Local definitions will be separated out into minimal, nested blocks of `ParsedLetrec`s and `ParsedLet`s (singleton, non-recursive local definitions) by dependency analysis (see Section 3.2). The parse scheme for `letrec`s is:

$$\mathcal{PE}\left[\begin{array}{ll} \texttt{let} & D_1 \\ & D_2 \\ & \ldots \\ & D_n \\ \texttt{in} & \\ & e\ \texttt{endlet} \end{array}\right] = \begin{array}{l} \texttt{new ParsedLetrec(}<\mathcal{PV}[D_1], \mathcal{PV}[D_2], \ldots, \mathcal{PV}[D_n]>\texttt{,} \\ \qquad\qquad\qquad\quad <\mathcal{PD}[D_1], \mathcal{PD}[D_2], \ldots, \mathcal{PD}[D_n]>\texttt{,} \\ \qquad\qquad\qquad\quad \mathcal{PE}[e]\texttt{)} \\ \\ \textbf{where} \quad \begin{array}{lcl} \mathcal{PD}[id\ x_1\ \ldots\ x_n\ \texttt{=}\ e] & = & \mathcal{PE}[\backslash x_1\ \ldots \backslash\ x_n\ e] \\ \mathcal{PV}[id\ x_1\ \ldots\ x_n\ \texttt{=}\ e] & = & \mathcal{PE}[id] \end{array} \end{array}$$

The constructor function of `ParsedLetrec` takes an array containing the variables defined (which may be functions), an array containing the corresponding definitions and the body of the expression.

## 3.2 Dependency Analysis

Dependency analysis is a means where we split a block of *potentially* mutually-recursive local definitions into nested blocks of minimally mutually-recursive `letrec`s and simple, non-recursive `let`s. For instance, if we have the following skeleton (due to [8]):

```
letrec
    a = ...;
    b = ...a...;
    c = ...b...h...c...;
    d = ...c...;
    f = ...g...h...a...;
    g = ...f...;
    h = ...g...;
in
    ...
```

Then, after dependency analysis, we have the expression:

```
let
    a = ...;
in let
    b = ...a...;
in letrec
    f = ...g...h...a...;
    g = ...f...;
    h = ...g...;
in letrec
    c = ...b...h...c...;
    d = ...c...;
in
    ...
```

The definition of `b` is dependent on that of `a` and must come after it. Both these are non-recursive definitions that are not dependent on the other variables in the block and so can be defined in simple `let`s. The other variables form two blocks, the variables of each being dependent on the others (either directly or indirectly) in the block and so form `letrec`s. The dependence of `c` on `h` dictates the ordering of the blocks.

The reasons for doing dependency analysis are threefold:

- Simple `let` definitions are easier to compile than `letrec`s (see Section 4.4.11)
- When lambda-lifting a `letrec` block, we need to include *all* free variables of the definitions in the `letrec` as all the definitions are potentially mutually-recursive and hence the free variables may need to be passed as parameters. Splitting the block into groups of definitions that *are* all mutually recursive minimises the number of extra parameters needed.
- Dependency analysis increases the efficiency of operations such as type-checking and strictness analysis.

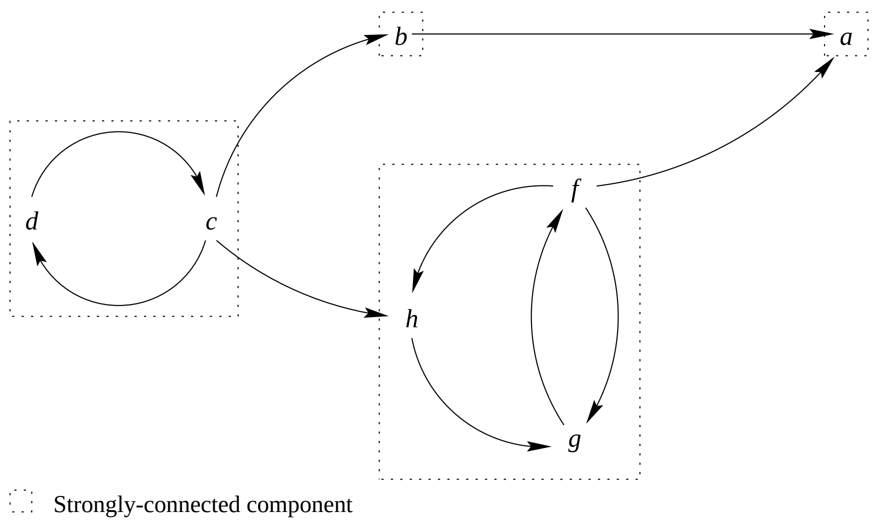


Figure 2: Example Dependency graph

Dependency analysis is a tricky and involved operation. We have the following steps:

1. Rename all variables so that they are unique. This removes a lot of potential headaches. We define a class `Substitutions` to store the details of the renaming. This class will also come in use when we come to do lambda-lifting (see Section 3.3).

2. Annotate each definition in the `letrec` with its free variables. This is accomplished by having a `free_vars` field in the `ParsedExpr` class (and its subclasses) which is a `Vector` holding the free variables. Each `ParsedExpr` subclass has a method for working out its free variables.

3. Form the dependency graph for the local variable definitions. There is a (uni-directional) path from one variable to another if the definition of the first variable contains a free instance of the second variable (see Figure 2). A directed graph class (`Graph`) was created to deal with this.

4. Find the strongly-connected components of the graph and perform a topological sort on these components [1]. These components form the minimally mutually-recursive blocks and the sort indicates how we are to nest the blocks. In our example, the sorted strongly-connected components are: {`c, d`}, {`b`}, {`f, g, h`}, {`a`}[2]. This means the definition of `c` and `d` can be enclosed in that of `b`, which can be enclosed in that of `f`, `g` and `g`, which can be enclosed in that of `a`. The components are returned in our implementation as a `Vector` of `Vectors` [10].

5. Any singleton components which don't point to themselves, i. e. are non-recursive, can be formed into a `let` expression. The rest are formed into `letrec` blocks. We now have the result given above.

The implementation is a straightforward implementation of this. We must, of course, perform dependency analysis on each of the definitions and the body of a `letrec` before performing dependency analysis on the `letrec` block itself. The actual method has prototype:

```
public ParsedExpr dependencyAnalysis(FunctionList functions)
```

We need the list of defined functions so that we don't treat them as free variables. The method `dependencyAnalysis` returns the result of doing dependency analysis on a `ParsedExpr`.

## 3.3 Lambda-Lifting

Lambda-lifting removes local function definitions and $\lambda$-expressions by replacing them with supercombinators and standard function calls.

[2]Since the dependencies between the strongly-connected components may only give way to a partial ordering we may have a number of possible, equally valid, sorts. Since the components {`b`} and {`f, g, h`} are independent of each other, an alternative sort is {`c, d`} {`f, g, h`}, {`b`}, {`a`}.

### 3.3.1 Lifting $\lambda$-expressions

Given a $\lambda$-expression we want to lift it out into a separate supercombinator and replace the $\lambda$-expression with a function call. We lift out nested lambdas *en bloc*, i. e. given an expression of the form $\lambda x_1 \ldots \lambda x_n.E$ (where $E$ is *not* a $\lambda$-expression) we form the supercombinator with arguments $x_1, \ldots, x_n$ directly (plus any free variables of $E$) rather than lifting each $\lambda$ separately. We have the following algorithm:

1. Create an argument `Vector` and add to it the variables of the $\lambda$-expression and any leading $\lambda$s.
2. Lambda-lift the body of the innermost leading $\lambda$-expression.
3. Recompute the free variables of this body after lambda-lifting and add them to the *beginning* of the argument `Vector`.
4. Create a supercombinator with this argument array and whose reduction rule is the lambda-lifted body. Add this supercombinator to the function list.
5. The result of lambda-lifting this expression is the application of the newly-generated supercombinators to the free variables calculated in 3 (making sure that the order of the variables is consistent).

### 3.3.2 Lifting local function definitions

We choose the Johnsson style of lambda-lifting [9] as it leads to supercombinators with fewer arguments than the normal method, which leads to recursive functions that take references to themselves as extra parameters. This, at the current stage, is our main reason for doing dependency analysis.

Local function definitions may occur in a `let` of a `letrec` and are indicated by the presence of leading lambdas in their definition[3]. As with $\lambda$-expressions we lift out all leading lambdas at once. We have the following algorithm for a `let` or `letrec` expression $L$:

1. For each local definition:
   (a) If we have a local function definition (i. e. leading lambdas are present), $f = D$, say. Then create a new supercombinator with *all* the free variables of the `let` or `letrec` block as parameters and $D$ as its reduction rule. Absorb the leading-lambdas of this new supercombinator as extra parameters and lambda-lift this new supercombinator. Replace all occurrences of $f$ in $L$ by the application of the new supercombinator to the free variables of the block.
   (b) If we have a local variable definition, then simply lambda-lift the definition.
2. Lambda-lift the body of $L$.

The substitutions of the occurrences of local variables by function applications are done all at the same time rather than singly, and make use of the `Substitutions` class.

# 4 Compilation

After parsing, dependency analysis and lambda-lifting, we now have a simplified graph in memory which can now be compiled. Compilation will 'flatten' the graph into Java byte code which will, when run, reconstruct the graph and reduce it to WHNF, i. e. evaluate the program that the graph represents.

We need to generate the class corresponding to a Ginger program, with each supercombinator compiled to a static method of that class. If the Ginger program defines the function `main` then we also need to include a `main` method in our target class (we may not wish to include a `main` function in a Ginger program that contains library functions, for example).

We do not create a Java class file directly, but rather target Jasmin assembly language [7]. This language is very similar to the byte code used by the Java Virtual Machine, but is easier to program in as it deals with such things as the Java constant pool (where all constants and object references as such) and calculating offsets for jumps automatically.

Our Ginger program, in the file *prog*`.g`, is compiled into a Jasmin file, *prog*`.j`, which is compiled into a Java class file, *prog*`.class` by the Jasmin assembler. If the Ginger program had `main` defined then

[3] Remember, that local function definitions are converted to the binding of a variable to a $\lambda$-expression — see Section 3.1.

| Scheme | Java method | Brief Description |
|---|---|---|
| $\mathcal{P}$ | `Code.compile` | Compile a program (a list of supercombinators). |
| $\mathcal{F}$ | `ParsedSuperComb.compile` | Compiles a supercombinator. |
| $\mathcal{R}$ | `ParsedExpr.r_compile` | Compiles the definition of a supercombinator. |
| $\mathcal{C}$ | `ParsedExpr.compile` | Compiles an expression (default method). |
| $\mathcal{E}$ | `ParsedExpr.e_compile` | Compiles a subexpression that is part of a *strict* expression. Places an `eval` instruction after the compilation. |
| $\mathcal{CL}$ | `ParsedLetrec.compileDefs` | Compile the local definitions of a `letrec` expression. |
| $\mathcal{A}$ | `ParsedLetrec.allocate` | Allocate the registers used by a `letrec` |
| $\mathcal{M}$ | `Code.main` | Create the `main` method of the target class. |
| $\mathcal{I}$ | — | Initialise the run-time function list. |

Table 2: The compilation schemes.

this class file is executable in the normal way. We can remove the `.j` file after compiling if we desire. Certainly this is desirable as it is roughly 4 times bigger than the corresponding `.class` file, though at present its existence is useful for debugging purposes.

We have a number of compile schemes, based on those described in [8]. A brief summary of them can be found in Table 2. The main schemes are:

- $\mathcal{C}$ which is the default scheme for compiling an expression.
- $\mathcal{R}$ which is the initial scheme to compile the RHS of a supercombinator definition. Using this we can avoid building graphs by exploiting tail recursion, i. e. when the reduction rule of a supercombinator is a call to a function which has the correct number of arguments present, and the strictness of known functions.
- $\mathcal{E}$ which is used to compile *and evaluate* an expression which is part of a strict super-expression.

We shall give details of each of the schemes below.

## 4.1 Abbreviations and Types

The JVM uses a coding scheme to represent the various Java types. These codes can be seen in Table 3.

| Code | Java type | Code | Java type |
|---|---|---|---|
| `B` | `byte` (8-bit integer) | `J` | `long` (64-bit integer) |
| `C` | `char` | `S` | `short` (16-bit integer) |
| `D` | `double` (64-bit floating point) | `Z` | `boolean` |
| `F` | `float` (32-bit floating point) | `L`*Class*`;` | The java class *Class*. |
| `I` | `int` (32-bit integer) | `[`*type* | An array of *type*. |

Table 3: JVM types

For brevity, the abbreviations detailed in Table 4 are used in the compilation schemes. The full names are required by Jasmin. `fp.gingerc` is the package where all the classes associated with the compiler reside.

## 4.2 Compiling a Program

If we treat our Ginger program as a list of supercombinators, $S_1, \ldots, S_n$ then our primary compilation scheme, $\mathcal{P}$, is: where $\phi$ is a list of all the defined functions (built up during parsing and lambda-lifting),

| Abbreviation | Jasmin Object |
|---|---|
| *Node* | `fp/gingerc/Node` |
| *Cons* | `fp/gingerc/Cons` |
| *EmptyList* | `fp/gingerc/EmptyList` |
| *Function* | `fp/gingerc/Function` |
| *Object* | `java/lang/Object` |
| *String* | `java/lang/String` |
| *Method* | `java/lang/reflect/Method` |

Table 4: The abbreviation used in our compile schemes.

$$\begin{array}{rcl} \mathcal{P}[S_1,\ldots,S_n]\ \phi\ c\ m & = & \texttt{.class public}\ c \\ & & \texttt{.super}\ \ Object \\ & & \mathcal{F}[S_1]\ \phi \\ & & \ldots \\ & & \mathcal{F}[S_n]\ \phi \\ & & \mathcal{M}\ \phi\ m \end{array}$$

$c$ is the name of the class to create, and $m$ is a boolean determining whether we create a `main` method or not.

The `.class public` command declares a new class and the `.super` declares its superclass. We then compile each of our supercombinators using the $\mathcal{F}$ scheme (see Section 4.3) and finally create a main method, using $\mathcal{M}$ (see Section 4.5), if we need one.

## 4.3 Compiling Supercombinators

To compile a supercombinator we have to create the header for a static method with the appropriate number of parameters and compile the supercombinators definition.

$$\begin{array}{rcl} \mathcal{F}[f\ x_1\ x_2\ \ldots\ x_n = E]\ \phi & = & \texttt{.method public static}\ f\texttt{(}\underbrace{\texttt{L}Node\texttt{;}\ldots\texttt{L}Node\texttt{;}}_{n \text{ times}}\texttt{)L}Node\texttt{;} \\ & & \quad \mathcal{R}[E]\ [x_1 = 0,\ldots,x_n = n-1]\ \phi\ n\ 0 \\ & & \quad \texttt{areturn} \\ & & \texttt{.end method} \end{array}$$

This creates a public, static method which takes $n$ parameters of type `Node` and returns an object of this type (indicated by the `areturn`).
It thus remains to describe the $\mathcal{R}$, $\mathcal{C}$ and $\mathcal{E}$ compilation schemes. These takes as arguments:

- The expression to be compiled. The syntax of this is given in Section 4.4.1.
- An environment. The parameters of the function and its local variables are held in the method's local variables. The environment details where they can be found. The parameters of the method generated are stored in the local variables $0,\ldots,n-1$ where $n$ is the arity of the source supercombinator. Any local variables will be stored from local variable $n$ onwards. Thus we don't have to treat parameter and local variables separately.
- A function list. Holds the list of all defined functions and their fully qualified names (i. e. with the full name of their defining class prefixed to each function).
- The next free variable register.

- The next free label. This is used to generate unique labels when we compile `ifs`.

The expression compilation schemes will leave a reference to the graph it creates on top of the JVM stack, removing any intermediate results. This will be returned by the `areturn` instruction generated by the $\mathcal{F}$ scheme.

## 4.4 Compiling Expressions

### 4.4.1 Expression BNF

After parsing, dependency analysis and lambda-lifting, our expressions have the following syntax:

$$
\begin{array}{lcl}
< \mathit{expr} > & ::= & < \mathit{integer} > \\
& | & < \mathit{boolean} > \\
& | & < \mathit{float} > \\
& | & < \mathit{character} > \\
& | & < \mathit{string} > \\
& | & < \mathit{identifier} > \\
& | & \texttt{[]} \\
& | & (< \mathit{expr} > \texttt{:} < \mathit{expr} >) \\
& | & (< \mathit{expr} > < \mathit{expr} >) \\
& | & \texttt{if} \;\; < \mathit{expr} > \;\; \texttt{then} \;\; < \mathit{expr} > \texttt{else} \;\; < \mathit{expr} > \;\; \texttt{endif} \\
& | & \texttt{let} \;\; < \mathit{identifier} > \;\; \texttt{=} \;\; < \mathit{expr} > \;\; \texttt{in} \;\; < \mathit{expr} > \;\; \texttt{endlet} \\
& | & \texttt{letrec} \\
& & \quad < \mathit{identifier} > \;\; \texttt{=} \;\; < \mathit{expr} > \\
& & \quad \ldots \\
& & \quad < \mathit{identifier} > \;\; \texttt{=} \;\; < \mathit{expr} > \\
& & \texttt{in} \;\; < \mathit{expr} > \;\; \texttt{endletrec}
\end{array}
$$

### 4.4.2 Compiling Integers

We need to create a new integer node and push it on the stack. There are no special cases for the $\mathcal{R}$ or $\mathcal{E}$ schemes as the resulting node will be fully evaluated.

$$
\begin{array}{rcl}
\mathcal{R}[i] \; \rho \; \phi \; k \; l & = & \mathcal{E}[i] \; \rho \; \phi \; k \; l \\
& = & \mathcal{C}[i] \; \rho \; \phi \; k \; l \\
& = & \texttt{new} \; \mathit{Node} \\
& & \texttt{dup} \\
& & L(i)
\end{array}
$$

The instruction `new` *Node* creates a new instance of a `Node` and leaves it on top of the (JVM) stack. `dup` duplicates the top element of the stack, leaving the duplicate on top of the stack. We need this duplicate because during initialisation of an object, the JVM pops a reference to that object off the stack. If we didn't have a copy of that object, we wouldn't be able to refer to the instance later on.

$L$ deals with loading the actual integer onto the stack and initialising the node. There are a number of ways we can push an integer on the stack, ranging in efficiency, depending on the size of the integer in question.. There are dedicated instructions for the constants $-1$ and $1, \ldots, 5$. These instructions are the fastest way to load these constants on the stack (they only take up one byte of byte-code, too).

$$
\begin{array}{rcll}
L(i) & = & \texttt{iconst\_}i & \\
& & \texttt{invokespecial} \; \mathit{Node}\texttt{/<init>(I)V} & , i \in \{0, \ldots, 5\} \\
& = & \texttt{iconst\_m1} & \\
& & \texttt{invokespecial} \; \mathit{Node}\texttt{/<init>(I)V} & , i = -1
\end{array}
$$

The `invokespecial` *Node*`/<init>(I)V` instruction initialises the `Node` instance by popping a reference to it and an `int` (specified by the '`I`') off the stack and calling the appropriate constructor function (see

Section 2.2). The '`V`' at the end of the instruction indicates that this operation returns the void type, i. e. doesn't return anything.

There are other instructions depending on whether the integer is in fact a `byte`, a `short` an `int` or a `long` (or rather, small enough to fit in the required type). Note that the JVM will automatically cast `byte`s and `short`s to `int`s as support for the former two types is limited at the JVM level. We have:

$$
\begin{array}{rcll}
L(i) & = & \texttt{bipush}\ i & \\
 & & \texttt{invokespecial}\ \mathit{Node}/\texttt{<init>(I)V} & ,\mathit{byte}\ i \\
 & = & \texttt{sipush}\ i & \\
 & & \texttt{invokespecial}\ \mathit{Node}/\texttt{<init>(I)V} & ,\mathit{short}\ i \\
 & = & \texttt{ldc}\ i & \\
 & & \texttt{invokespecial}\ \mathit{Node}/\texttt{<init>(I)V} & ,\mathit{int}\ i \\
 & = & \texttt{ldc\_2w}\ i & \\
 & & \texttt{invokespecial}\ \mathit{Node}/\texttt{<init>(J)V} & ,\mathit{long}\ i
\end{array}
$$

`bipush` is more efficient that `sipush` which is more efficient than `ldc`, etc. Note that in the last case, we have an integer that is too big to fit in the 32-bit cells that the JVM uses for its stack elements, so we have to split it over two cells (using the `ldc_2w` instruction). This is reflected in the `invokespecial` *Node*/`<init>(J)V` which initialises the node using a `long` (indicated by the '`J`'). This use of extra space (as well as speed reasons) is why we prefer to compile using `int`s if possible.

### 4.4.3 Compiling Floats

As with the integer case, we have a number of possible compile methods, depending on the value of the float in question. There are no special cases for the $\mathcal{R}$ or $\mathcal{E}$ schemes as the resulting node will be fully evaluated.

$$
\begin{array}{rcl}
\mathcal{R}[x]\ \rho\ \phi\ k\ l & = & \mathcal{E}[x]\ \rho\ \phi\ k\ l \\
 & = & \mathcal{C}[x]\ \rho\ \phi\ k\ l \\
 & = & \texttt{new}\ \mathit{Node} \\
 & & \texttt{dup} \\
 & & F(x) \\
 & & \textbf{where} \\
 & & \begin{array}{rcll}
 F(x) & = & \texttt{fconst\_0} & \\
 & & \texttt{invokespecial}\ \mathit{Node}/\texttt{<init>(F)V} & ,x = 0.0 \\
 & = & \texttt{fconst\_1} & \\
 & & \texttt{invokespecial}\ \mathit{Node}/\texttt{<init>(F)V} & ,x = 1.0 \\
 & = & \texttt{fconst\_2} & \\
 & & \texttt{invokespecial}\ \mathit{Node}/\texttt{<init>(F)V} & ,x = 2.0 \\
 & = & \texttt{ldc}\ x & \\
 & & \texttt{invokespecial}\ \mathit{Node}/\texttt{<init>(F)V} & ,\mathit{float}\ x \\
 & = & \texttt{ldc\_2w}\ x & \\
 & & \texttt{invokespecial}\ \mathit{Node}/\texttt{<init>(D)V} & ,\mathit{double}\ x
 \end{array}
\end{array}
$$

As with `long`/`int` case, `double`s take twice as much space as `float`s and must be dealt with accordingly. Again we prefer to compile using the single-precision `float`s rather than the double-precision `double`s if possible.

### 4.4.4 Compiling Characters

We have to simply load the character onto the stack. There are no special cases for the $\mathcal{R}$ or $\mathcal{E}$ schemes as the resulting node will be fully evaluated.

$$
\begin{array}{rcl}
\mathcal{R}[c]\ \rho\ \phi\ k\ l & = & \mathcal{E}[c]\ \rho\ \phi\ k\ l \\
 & = & \mathcal{C}[c]\ \rho\ \phi\ k\ l \\
 & = & \texttt{new}\ \mathit{Node} \\
 & & \texttt{dup} \\
 & & \texttt{ldc}\ c \\
 & & \texttt{invokespecial}\ \mathit{Node}/\texttt{<init>(C)V}
\end{array}
$$

### 4.4.5 Compiling Strings

In this case we simply need to load the string on the stack. There are no special cases for the $\mathcal{R}$ or $\mathcal{E}$ schemes as the resulting node will be fully evaluated.

$$\begin{aligned}
\mathcal{R}[s]\,\rho\,\phi\,k\,l &= \mathcal{E}[s]\,\rho\,\phi\,k\,l \\
&= \mathcal{C}[s]\,\rho\,\phi\,k\,l \\
&= \texttt{new}\ \mathit{Node} \\
&\phantom{=}\ \texttt{dup} \\
&\phantom{=}\ \texttt{ldc "}s\texttt{"} \\
&\phantom{=}\ \texttt{invokespecial}\ \mathit{Node}/\texttt{<init>(L}\mathit{Object}\texttt{;)V}
\end{aligned}$$

Note the need for the quote marks.

The `invokespecial` *Node*`/<init>(L`*Object*`;)V` instruction constructs a `Node` containing an object. As well as constructing nodes containing strings (represented as objects in Java), it is also used to create nodes containing cons nodes.

### 4.4.6 Compiling Booleans and the Empty List

The booleans, `true` and `false`, and the empty list, `[]`, form three constants in our language. Their compilation scheme is unvarying, moreover because of referential transparency and our use of a boxed implementation of nodes, we only need to have *one* instance of these constants during the lifetime of the program. This saves time and space when both compiling and running the program. We store instances of these constants in the class `Function`, *viz*:

```
public static final Node empty_list = new Node(new EmptyList());
public static final Node true_node  = new Node(true);
public static final Node false_node = new Node(false);
```

This class is also the home of our run-time function list (see below). Compiling one of these constants is just a case of getting a reference to the appropriate constant. As the constants are already in WHNF all three compilation schemes are identical.

$$\begin{aligned}
\mathcal{R}[\texttt{[]}]\,\rho\,\phi\,k\,l &= \mathcal{E}[\texttt{[]}]\,\rho\,\phi\,k\,l \\
&= \mathcal{C}[\texttt{[]}]\,\rho\,\phi\,k\,l \\
&= \texttt{getstatic}\ \mathit{Function}/\texttt{empty_list L}\mathit{Node}\texttt{;}
\end{aligned}$$

and similarly for the two booleans. The `getstatic` is used to read a static field. Its first argument is the full name of the required field; its second argument is the type of the field.

### 4.4.7 Compiling Identifiers

An identifier represents either a function or a variable, both of which must be treated separately. If an identifier has a corresponding register defined in the environment then it must be a variable; if it is defined in the function list then we have a function; if it is defined in neither then we have an error.

Compiling variables is easy: we just load the reference contained in the appropriate local variable, *viz*:

$$\mathcal{C}[x]\,\rho\,\phi\,k\,l \quad = \quad \texttt{aload}\ \rho(x)$$

Compiling functions is a lot more involved. We need to put the appropriate `Method` on top of the stack. This involves using the method `getMethod` in `java.lang.Class`, for which we need the `Class` object reflecting the class of the function we are compiling (the method `forName` will return a `Class` object represented by a string), an array representing the arguments of the function[4] and its name. Since we may wish to compile the same function many times, we can see that there may be a lot of repeated work.

We instead choose to keep a run-time list of compiled functions (held as a Hash table in the `Function` class) which a compiled Ginger program can consult when it is running. This is similar to the idea of

[4]Since Java methods are overloaded, we cannot just refer to a function just by its name.

using fields to store pre-compiled constants as detailed above. The creation of a function node thus involves loading the name of the function on the stack[5] and looking up this name in our list, *viz*:

$$\begin{aligned}\mathcal{C}[f]\,\rho\,\phi\,k\,l &= \texttt{ldc}\ \phi(f)\\ &\phantom{=}\ \texttt{invokestatic}\ \mathit{Function}\texttt{/lookup(L}\mathit{String}\texttt{;)L}\mathit{Method}\texttt{;}\end{aligned}$$

The function call $\phi(f)$ returns the fully qualified name of $f$.

When the identifier appears on its own in the RHS of a supercombinator definition, we need to evaluate it before updating the root of the redex (see Section 5.2) to preserve laziness. Similarly if the identifier occurs as part of a strict expression then we need to evaluate it. We have:

$$\begin{aligned}\mathcal{R}[id]\,\rho\,\phi\,k\,l &= \mathcal{E}[id]\,\rho\,\phi\,k\,l\\ &= \mathcal{C}[id]\,\rho\,\phi\,k\,l\\ &\phantom{=}\ \texttt{invokevirtual}\ \mathit{Node}\texttt{/eval()L}\mathit{Node}\texttt{;}\end{aligned}$$

#### 4.4.8 Compiling Cons

We compile the head and tail of the list, make a `Cons` node and finally box it in a `Node` object. There are no special cases for the $\mathcal{R}$ or $\mathcal{E}$ schemes as the resulting node will be in WHNF. We compile the tail of the list first.

$$\begin{aligned}\mathcal{R}[h:t]\,\rho\,\phi\,k\,l &= \mathcal{E}[h:t]\,\rho\,\phi\,k\,l\\ &= \mathcal{C}[h:t]\,\rho\,\phi\,k\,l\\ &= \texttt{new}\ \mathit{Node}\\ &\phantom{=}\ \texttt{dup}\\ &\phantom{=}\ \texttt{new}\ \mathit{Cons}\\ &\phantom{=}\ \texttt{dup}\\ &\phantom{=}\ \mathcal{C}[t]\,\rho\,\phi\,k\,l\\ &\phantom{=}\ \mathcal{C}[h]\,\rho\,\phi\,k\,l\\ &\phantom{=}\ \texttt{invokespecial}\ \mathit{Cons}\texttt{/<init>(L}\mathit{Node}\texttt{;L}\mathit{Node}\texttt{;)V}\\ &\phantom{=}\ \texttt{invokespecial}\ \mathit{Node}\texttt{/<init>(L}\mathit{Object}\texttt{;)V}\end{aligned}$$

#### 4.4.9 Compiling Applications

Given an application $E_1\ E_2$ we have to compile each of the subexpressions, form them into an application node and box them into a `Node` object. The constructor method of `Node` taking two parameters of type `Object` creates a boxed application from its arguments. The default for $\mathcal{R}$ and $\mathcal{C}$ is:

$$\begin{aligned}\mathcal{R}[E_1\ E_2]\,\phi\,\rho\,k\,l &= \mathcal{C}[E_1\ E_2]\,\phi\,\rho\,k\,l\\ &= \texttt{new}\ \mathit{Node}\\ &\phantom{=}\ \texttt{dup}\\ &\phantom{=}\ \mathcal{C}[E_2]\,\phi\,\rho\,k\,l\\ &\phantom{=}\ \mathcal{C}[E_1]\,\phi\,\rho\,k\,l\\ &\phantom{=}\ \texttt{invokespecial}\ \mathit{Node}\texttt{/<init>(L}\mathit{Object}\texttt{;L}\mathit{Object}\texttt{;)V}\end{aligned}$$

For $\mathcal{E}$, we can evaluate the application once we have constructed it:

$$\begin{aligned}\mathcal{E}[E_1\ E_2]\,\phi\,\rho\,k\,l &= \mathcal{C}[E_1\ E_2]\,\phi\,\rho\,k\,l\\ &\phantom{=}\ \texttt{invokevirtual}\ \mathit{Node}\texttt{/eval()L}\mathit{Node}\texttt{;}\end{aligned}$$

If the application is known to be a strict application (in all its arguments) with all its arguments present then we can evaluate its arguments and call it directly. At present, the only known strict functions

[5] Ginger functions are *not* overloaded and hence can be referred to uniquely by name.

are the primitives loaded from the file `StrictPrimitives` (these are basically the strict operators from Table 4) and those used in the antecedent of a conditional.

$$\begin{aligned}
\mathcal{R}[f\ E_1\ \ldots\ E_n]\ \phi\ \rho\ k\ l &= \mathcal{E}[f\ E_1\ \ldots\ E_n]\ \phi\ \rho\ k\ l \\
&= \mathcal{C}[f\ E_1\ \ldots\ E_n]\ \phi\ \rho\ k\ l \\
&= \mathcal{E}[E_1]\ \phi\ \rho\ k\ l \\
&\quad \ldots \\
&\quad \mathcal{E}[E_n]\ \phi\ \rho\ k\ l \\
&\quad \texttt{invokestatic}\ \phi(f)\texttt{/(}\underbrace{\texttt{L}\mathit{Node}\texttt{;}\ldots\texttt{L}\mathit{Node}\texttt{;}}_{n\text{ times}}\texttt{)L}\mathit{Node}\texttt{;} \\
&\quad \textbf{where } f \text{ is strict and of arity } n
\end{aligned}$$

Each argument $E_i$ of the application is compiled and evaluated *via* $\mathcal{E}$. The $n$ arguments of $f$ are then on the stack in the required order. The command `invokestatic` calls the functions with the appropriate number of elements from the stack and leaves the answer on top of the stack.

If we have a tail recursion then we can compile the application directly:

$$\begin{aligned}
\mathcal{R}[f\ E_1\ \ldots\ E_n]\ \phi\ \rho\ k\ l &= \mathcal{C}[E_1]\ \phi\ \rho\ k\ l \\
&\quad \ldots \\
&\quad \mathcal{C}[E_n]\ \phi\ \rho\ k\ l \\
&\quad \texttt{invokestatic}\ \phi(f)\texttt{/(}\underbrace{\texttt{L}\mathit{Node}\texttt{;}\ldots\texttt{L}\mathit{Node}\texttt{;}}_{n\text{ times}}\texttt{)L}\mathit{Node}\texttt{;} \\
&\quad \textbf{where } f \text{ is of arity } n
\end{aligned}$$

### 4.4.10 Compiling `ifs`

We need to compile the antecedent (using the $\mathcal{E}$ scheme as it is always evaluated) and branch accordingly. The $l$ parameter is there to ensure unique labels and counts the nesting of multiple `ifs`. Note that we could have `ifs` inside the antecedent as well so we have to compile this with the new label, too.

$$\begin{aligned}
\mathcal{C}[\ \texttt{if}\ A\ \texttt{then}\ T\ \texttt{else}\ F\ \texttt{endif}\ ]\ \rho\ \phi\ k\ l &= \mathcal{E}[A]\ \rho\ \phi\ k\ (l+1) \\
&\quad \texttt{getfield}\ \mathit{Node}\texttt{/contents L}\mathit{Object}\texttt{;} \\
&\quad \texttt{checkcast}\ \mathit{Boolean} \\
&\quad \texttt{invokevirtual}\ \mathit{Boolean}\texttt{/booleanValue()Z} \\
&\quad \texttt{ifeq FALSE_}l \\
&\quad \texttt{TRUE_}l\texttt{:} \\
&\qquad \mathcal{C}[T]\ \rho\ \phi\ k\ (l+1) \\
&\qquad \texttt{goto ENDIF_}l \\
&\quad \texttt{FALSE_}l\texttt{:} \\
&\qquad \mathcal{C}[F]\ \rho\ \phi\ k\ (l+1) \\
&\quad \texttt{ENDIF_}l\texttt{:}
\end{aligned}$$

The `checkcast` instruction ensures the legitimacy of the cast needed to extract the boolean value field from what is ostensibly an `Object` but, in a correctly typed program, should be a `Boolean`.

In the $\mathcal{R}$ and the $\mathcal{E}$ schemes we can propagate the appropriate scheme through the branches. For $\mathcal{R}$ we have:

$$\begin{aligned}
\mathcal{R}[\ \texttt{if}\ A\ \texttt{then}\ T\ \texttt{else}\ F\ \texttt{endif}\ ]\ \rho\ \phi\ k\ l &= \mathcal{E}[A]\ \rho\ \phi\ k\ (l+1) \\
&\quad \texttt{getfield}\ \mathit{Node}\texttt{/contents L}\mathit{Object}\texttt{;} \\
&\quad \texttt{checkcast}\ \mathit{Boolean} \\
&\quad \texttt{invokevirtual}\ \mathit{Boolean}\texttt{/booleanValue()Z} \\
&\quad \texttt{ifeq FALSE_}l \\
&\quad \texttt{TRUE_}l\texttt{:}
\end{aligned}$$

$$\begin{array}{l} \quad \mathcal{R}[T]\ \rho\ \phi\ k\ (l+1) \\ \quad \texttt{goto ENDIF\_}l \\ \texttt{FALSE\_}l\texttt{:} \\ \quad \mathcal{R}[F]\ \rho\ \phi\ k\ (l+1) \\ \texttt{ENDIF\_}l\texttt{:} \end{array}$$

and similarly for $\mathcal{E}$.

### 4.4.11 Compiling lets

We have two cases: that of a single, non-recursive definition; and that of a block of mutually-recursive definitions. The single, non-recursive case is fairly easy. After compiling the definition of the local variable, we store it in the next free variable and then compile the body of the `let` with the environment and the next free variable updated accordingly.

$$\begin{array}{rcl} \mathcal{C}[\ \texttt{let}\ x = E_d\ \texttt{in}\ E\ \texttt{endlet}\ ]\ \rho\ k\ l & = & \mathcal{C}[E_d]\ \rho\ \phi\ k\ l \\ & & \mathit{astore}(k) \\ & & \mathcal{C}[E]\ \rho[x=k]\ \phi\ (k+1)\ l \end{array}$$

The function *astore* selects the most efficient way of storing a reference in the specified local variable.

As with the conditional, we can propagate the $\mathcal{R}$ and the $\mathcal{E}$ into the body of the `let` (but *not* the definitions as they may never be evaluated). For $\mathcal{R}$ we have:

$$\begin{array}{rcl} \mathcal{R}[\ \texttt{let}\ x = E_d\ \texttt{in}\ E\ \texttt{endlet}\ ]\ \rho\ k\ l & = & \mathcal{C}[E_d]\ \rho\ \phi\ k\ l \\ & & \mathit{astore}(k) \\ & & \mathcal{R}[E]\ \rho[x=k]\ \phi\ (k+1)\ l \end{array}$$

and similarly for $\mathcal{E}$.

The case of a block of mutually-recursive definitions is more complex. We need to compile the definitions of the variables in an environment which contains the local registers where all the variables are stored. Moreover these registers must contain the correct `Node`. We thus need to create 'place-holder' nodes, one for each variable defined, which will be filled in when the appropriate definition is compiled.

$$\begin{array}{rcl} \mathcal{C}[\ \texttt{let}\ D\ \texttt{in}\ E\ \texttt{endlet}\ ]\ \rho\ k\ l & = & \mathcal{A}(D)\ k \\ & & \mathcal{CL}(D)\ \rho'\ \phi\ k'\ l \\ & & \mathcal{C}(E)\ \rho'\ \phi\ k'\ l \\ & & \texttt{where}\ (\rho', k') = X[D]\ \rho\ k \end{array}$$

Where the function $X$ (defined below) updates the environment. Again we can propagate the $\mathcal{R}$ and the $\mathcal{E}$ into the body of the `letrec`. For $\mathcal{R}$ we have:

$$\begin{array}{rcl} \mathcal{C}[\ \texttt{let}\ D\ \texttt{in}\ E\ \texttt{endlet}\ ]\ \rho\ k\ l & = & \mathcal{A}(D)\ k \\ & & \mathcal{CL}(D)\ \rho'\ \phi\ k'\ l \\ & & \mathcal{R}(E)\ \rho'\ \phi\ k'\ l \\ & & \texttt{where}\ (\rho', k') = X[D]\ \rho\ k \end{array}$$

and similarly for $\mathcal{E}$. The scheme $\mathcal{A}$ creates the place-holders for each of the nodes, *viz*:

$$\mathcal{A}\left[\begin{array}{l} x_1 = E_1 \\ x_2 = E_2 \\ \dots \\ x_n = E_n \end{array}\right] k = \begin{array}{l} \texttt{new } \mathit{Node} \\ \texttt{dup} \\ \texttt{invokespecial } \mathit{Node}\texttt{/<init>()V} \\ \mathit{astore}(k) \\ \texttt{new } \mathit{Node} \\ \texttt{dup} \\ \texttt{invokespecial } \mathit{Node}\texttt{/<init>()V} \\ \mathit{astore}(k+1) \\ \dots \\ \texttt{new } \mathit{Node} \\ \texttt{dup} \\ \texttt{invokespecial } \mathit{Node}\texttt{/<init>()V} \\ \mathit{astore}(k+n-1) \end{array}$$

The scheme $\mathcal{CL}$ compiles each definition and updates the appropriate node.

$$\mathcal{CL}\left[\begin{array}{l} x_1 = E_1 \\ x_2 = E_2 \\ \dots \\ x_n = E_n \end{array}\right] \rho\ \phi\ k\ l = \begin{array}{l} \mathit{aload}\ \rho(x_1) \\ \mathcal{C}[E_1]\ \rho\ \phi\ k\ l \\ \texttt{invokevirtual } \mathit{Node}\texttt{/update(L}\mathit{Node}\texttt{;)V} \\ \mathit{aload}\ \rho(x_2) \\ \mathcal{C}[E_2]\ \rho\ \phi\ k\ l \\ \texttt{invokevirtual } \mathit{Node}\texttt{/update(L}\mathit{Node}\texttt{;)V} \\ \dots \\ \mathit{aload}\ \rho(x_n) \\ \mathcal{C}[E_n]\ \rho\ \phi\ k\ l \\ \texttt{invokevirtual } \mathit{Node}\texttt{/update(L}\mathit{Node}\texttt{;)V} \end{array}$$

Finally, the function $X$ updates the environment and the next free variable:

$$X\left[\begin{array}{l} x_1 = E_1 \\ x_2 = E_2 \\ \dots \\ x_n = E_n \end{array}\right] \rho\ k = \left(\rho\left[\begin{array}{l} x_1 = k \\ x_2 = k+1 \\ \dots \\ x_n = k+n-1 \end{array}\right], k+n\right)$$

### 4.5 Creating the `main` method

The `main` method of the target class needs to do two things:

1. Initialise the run-time function list.
2. Evaluate the `main` supercombinator and print it.

For simplicity, we rename the `main` supercombinator `_main`. Evaluation thus involves loading the `Node` containing the `Method` that reflects `_main`, evaluating it (by passing it an `eval` message) and printing it (by passing the result a `print` message. We have the following scheme:

$$\begin{array}{lll} \mathcal{M}\ \phi\ \mathit{makemain} & = & \begin{array}{l} \texttt{.method public static main([L}\mathit{String}\texttt{;)V} \\ \quad \mathcal{I}\ \phi \\ \quad \texttt{ldc \_main} \\ \quad \texttt{invokestatic } \mathit{Function}\texttt{/lookup(L}\mathit{String}\texttt{;)L}\mathit{Node}\texttt{;} \\ \quad \texttt{invokevirtual } \mathit{Node}\texttt{/eval()L}\mathit{Node}\texttt{;} \\ \quad \texttt{invokevirtual } \mathit{Node}\texttt{/print()V} \\ \quad \texttt{return} \\ \texttt{.end method} \end{array} \quad , \textbf{ if } \mathit{makemain} \\ & = & \text{“”} \quad , \textbf{ otherwise} \end{array}$$

where we use “” to indicate that nothing is generated. The instruction `invokevirtual` invokes an instance method, and takes a reference to the instance in question as an implicit parameter.

The $\mathcal{I}$ scheme initialises the function list. We load the class, name and arity of each function defined and invoke the static method `Function.add`. This creates a `Node` containing a `Method` object which reflects the method specified by the parameters[6] and adds it to the runtime list.

$$
\begin{array}{rcl}
\mathcal{I}\{f_1, \ldots, f_n\} & = & \texttt{invokestatic}\ \textit{Function}\texttt{/initialise()V} \\
& & \texttt{ldc}\ \textit{class}(f_1) \\
& & \texttt{ldc}\ \textit{name}(f_1) \\
& & \textit{loadint}(\textit{arity}(f_1)) \\
& & \texttt{invokestatic}\ \textit{Function}\texttt{/add(L}\textit{String}\texttt{;L}\textit{String}\texttt{;I)V} \\
& & \ldots \\
& & \texttt{ldc}\ \textit{class}(f_n) \\
& & \texttt{ldc}\ \textit{name}(f_n) \\
& & \textit{loadint}(\textit{arity}(f_n)) \\
& & \texttt{invokestatic}\ \textit{Function}\texttt{/add(L}\textit{String}\texttt{;L}\textit{String}\texttt{;I)V}
\end{array}
$$

### 4.6 Implementation Details

The compilation schemes detailed above are mirrored almost exactly in our implementation. The compilation code is generated by a `compile`, `e_compile` and `r_compile` methods associated with each of the various `ParsedNode` subclasses (see Section 3.1). However, Jasmin methods require an upper bound placed on the stack used and the number of local variables (including the method's parameters which are also stored in local parameters). This can be predicted during compilation.

## 5 Evaluating, Unwinding and Printing the Graph

We have already seen the abstract prototypes for evaluating and unwinding the graph (these implement the `EVAL` and `UNWIND` G-Machine instructions). We now give implementations of them.

### 5.1 Evaluating Graph Nodes

Passing an `eval` message to a graph node causes it to be evaluated, i e. reduced to WHNF, the result of the reduction is returned. If the node is already in WHNF, i. e. an integer, boolean etc. or a non-CAF function node then the default (inherited) method is just to return the node. If we have an application node then we must begin unwinding. If we have a CAF then we need to reduce it by invoking the associated function. In this case we also need to update the root of the redex, the node itself, before returning the result (in the application case, the updating is dealt with in the `unwind` method, see below).

We have the following instance method of the class `Node`:

```
public Node eval() {
  if (contents instanceof App) {
    // create a new stack containing this and unwind the applications
    Stack s = new Stack();
    s.push(this);
    return unwind(s);
  }
  else if (contents instanceof Method &&
          ((Method) contents).getParameterTypes().length == 0) {
    // we have a CAF -- apply and evaluate
    Object[] args = new Object[0];
    Node n = (Node) ((Method)contents).invoke(null, args);
    update(n.eval());
    return this;
  }
```

[6] All the parameters of the methods generated have type `Node`, hence we only need to pass the arity. This is all the information needed by the `add` method to create the parameter array (an array of classes reflecting the parameter types) needed by the `getMethod` method in `Class`.

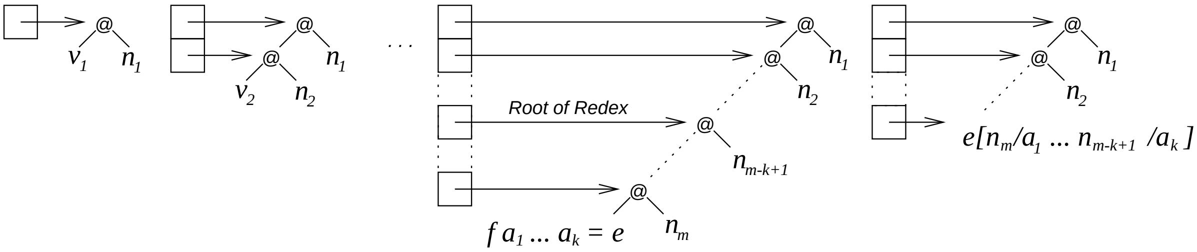


Figure 3: Unwinding an application node

```
  else
    // the node is in WHNF so return it
    return this;
}
```

Note that we have omitted the exception handlers for clarity.

The instance method `invoke` of a `Method` invokes the underlying method represented by the Method object, on the specified object (the first argument) with the specified parameters (held in an array in the second argument). If the underlying method is static then the first argument is ignored; this is so in our case and we use `null` as a dummy argument.

## 5.2 Unwinding Graph Nodes

Unwinding a graph node involves walking down the spine of a graph, storing the ribs in a stack, until we reach the base of the spine. At the base of the spine should, in a correctly-typed program, reside a function node. If this function node has enough arguments then we apply it and update the root of the redex; if not then the graph is already in WHNF and we just return its root (see Figure 5.2). The full code can be seen in Appendix C.

## 5.3 Printing Results

Printing a node is done mainly *via* the `toString` method, though printing a `Cons` node requires the evaluation of its head and tail (to WHNF) before printing.

## 5.4 Primitives

We have seen in Section 3.1.4 that our infix operators are converted into a prefix form. These operators, plus a handful of others — e. g. `strict` (force a strict application), `read` (read a file), etc. — form the primitives of our language. These primitives themselves are not built into the language but imported automatically from two separate class files — `StrictPrimitives` contains, not surprisingly, the strict primitives, while `Primitives` contains the non-strict ones. This is accomplished in the same manner that we import other Ginger functions (see below).

This class file is generated from a Java source. For example, the code for `_plus` is:

```
public static Node _plus(Node lhs, Node rhs) throws ClassCastException {
  Number l = (Number)lhs.eval().contents;
  Number r = (Number)rhs.eval().contents;

  if (l instanceof Long && r instanceof Long)
    return new Node(l.longValue() + r.longValue());
  else
    return new Node(l.doubleValue() + r.doubleValue());
}
```

The `_plus` function receives two nodes, the contents of which it has to add. Since `_plus` is strict in both arguments, it has to evaluate its parameters. The `contents` of the resulting `Nodes` should be castable to the Java class `Number` (a superclass of the Java classes `Long` and `Double`). Once we have done this, we need to decide whether we are doing an integer or a floating point addition. If both `contents` are

| Program | Interpreter (s) | | Compiler (s) | | |
|---|---|---|---|---|---|
| | Compile | Run | Compile | Run | Runtime Speedup |
| `thue 10` | 0.082 | 43.68 | 3.06 | 2.56 | 17.1 |
| `take 200 primes` | 0.114 | 45.97 | 2.41 | 13.45 | 3.4 |
| `hamming 100` | 0.23 | 8.15 | 3.7 | 0.64 | 12.73 |
| `edigits 25` | 0.169 | 94.04 | 3.01 | 2.00 | 47.2 |
| `nfib 20` | 0.188 | 60.63 | 2.15 | 6.97 | 8.7 |
| `square_root 2 200` | 0.542 | 23.76 | 3.18 | 0.81 | 29.3 |
| `quicksort` | 1.971 | 7.7 | 7.48 | 1.96 | 3.9 |
| `mergesort` | — | 13.55 | — | 1.32 | 10.26 |
| `bubblesort` | — | 273.44 | — | 11.86 | 23.1 |
| `insertionsort` | — | 32.14 | — | 1.49 | 21.57 |

Table 5: Comparing the compile- and run-times of the Ginger Interpreter and the Compiler. N.B. the four sorts are held in the same file and were evaluated on the result of `square_root 2 100.`

castable to `Long`s then we must be doing an integer add — we take the integer value of each, add them and return a new integer node. Otherwise we must be doing a floating point addition (this is the case for an integer plus a float, too) and we proceed accordingly. The `ClassCastException` is thrown if a type error occurs during execution. In a correctly-typed program this cannot occur, of course, but Java demands the presence of the `throws` expression and, at the moment, the Ginger compiler has no type-checker.

The other primitives are programmed similarly and as dictated by [5]. However, given the likely use of the compiler for exploration into fuzzy logic [6], we have chosen to overload the logic operators — `&`, `|` and `~` — so that they work on fuzzy values (i. e. floats), as well as booleans, using the min/max method [11]. We also choose to allow ordering comparison to be done on booleans.

### 5.5 Importing Functions

As mentioned above, we have a facility for importing functions, a process which is done automatically for the primitives and the Prelude[7] (the standard Ginger functions that aren't primitives) and is also available for manual import. We introduce a new primitive — `import` — to the Ginger syntax. This imports all public methods from a Java class file using the method `getDeclaredMethods`. This importing is in actual fact just an examination of the class file to get enough information about the methods contained therein so that we can add these functions to our function list (see Section 4.4.7) and initialise them during execution of the `main` method of our eventual target class[8]. The actual physical loading of the files is done by the Java class loader at compile time[9]. This import facility allows us to write the majority of the standard Ginger prelude functions in Ginger itself, far easier and neater than doing it in Java. We then import this compiled file whenever we compile a Ginger program.

## 6 Results

The times of compiling and running some of the standard programs in the Ginger test suite can be found in Table 5. The code for some of these programs can be seen in Appendix D. As we see, we achieve a speed-up of up to 47 times when compared to the interpreter. The idea behind compiling into supercombinators is that the absence of free variables allows us to substitute *all* the formal parameters of a function for their actual values in one step, rather than one at a time as in a combinator-based strategy, say. So the more functions of arity greater than one that a program has, the greater the speed-up achieved (when compared to the combinator-based Ginger interpreter). In simple terms, the more functions we have of arity 2 or greater, and the more they are used, the greater the speed-up achieved.

[7]The Prelude *isn't* imported if that is the file we are compiling!

[8]The `import` declarations are lost during compilation, and thus if an imported file needs to import another file, then this importing must be done at the top level.

[9]This is why Jasmin requires the package and class name every time it invokes a function, so that it knows what files it needs to import.

We also achieve greater speed-ups by directly compiling supercombinators which have all their arguments present (and so avoid building graph) and exploiting knowledge on strictness.

For instance, the `primes` program which has the smallest speed-up has the definition:

```
primes = map hd (iterate sieve [2..]);

sieve l =
    let
        p = hd l;
        xs = tl l;
    in
        filter (\x x % p ~= 0) xs
    endlet;

main = take 200 primes;
```

has only one (rather simple) function of arity greater than 1 (this will arise from lifting the lambda expression in the filter expression) and only two subexpression (the modulus and the inequality) which can take advantage of direct compilation. If we consider the `edigits` program (used to calculate the digits of e) defined as:

```
edigits n = if n <= 0 then inflist else take n inflist endif;

inflist = 2 : convert (repeat 1);

convert x =
    let
        xx = norm 2 (0 : map ((*) 10) x);
    in
        hd xx : convert (tl xx)
    endlet;

norm c lis =
    let
         d = hd lis; f = tl lis; e = hd f; x = tl f;
    in
        if c > 9 + e % c
        then d + e / c : ee % c : xx
        else d + ee / c : ee % c : xx
        endif
        where
            eee = norm (1 + c) (e : x);
            ee = hd eee;
            xx = tl eee;
        endwhere
    endlet;

main = edigits 25;
```

which had the highest speed-up, we find there is not only plenty of scope for direct compilation, with the many occurrences of arithmetical and list operations, but the binary `norm` function is recursive and these recursive calls increase the speed-up effect of multi-parameter substitution.

# 7 Conclusion

We have succeeded in producing a functional compiler for Ginger. This compiler has considerable speed-ups when compared to the original interpreter. The resulting Java byte-code can be run on any machine with a suitable Java interpreter. The absolute performance of the compiler should increase by an order of magnitude when stable Java Just-In-Time compilers are readily available.

The Java language is a useful one in which to implement a language, with several aspects of its object-oriented approach — namely inheritance and overloading — coming in very handy (though having to edit ten separate files to add a function to a class and its subclasses was a bit of a bind). The presence of classes for common data structures, e. g. stacks, hash tables and vectors, was most appreciated. On the down side, the lack of explicit first-class functions was a drawback, though we managed to emulate these using the appropriate Java classes.

The compiler will provide the base for further research into functional languages, in particular the use of fuzzy logic in a functional language.

# A Ginger BNF

Below can be found the BNF for Ginger. The BNF for the primitives (integers, etc.) follows the format used by Java, with `long`s and `double`s used for integers and floats [4].

$$
\begin{array}{rcl}
<\mathit{definition}> & ::= & <\mathit{identifier}><\mathit{identifier}>* \ \texttt{=}\ <\mathit{expr}>\ \texttt{;} \\
<\mathit{expr}> & ::= & <\mathit{integer}> \\
 & | & <\mathit{boolean}> \\
 & | & <\mathit{float}> \\
 & | & <\mathit{character}> \\
 & | & <\mathit{identifier}> \\
 & | & <\mathit{string}> \\
 & | & \texttt{[]} \\
 & | & \texttt{[}(<\mathit{expr}>\ \texttt{,})*<\mathit{expr}>\texttt{]} \\
 & | & \texttt{[}<\mathit{expr}>\texttt{..]} \\
 & | & \texttt{[}<\mathit{expr}>\texttt{..}<\mathit{expr}>\texttt{]} \\
 & | & \texttt{[}<\mathit{expr}>\texttt{,}<\mathit{expr}>\texttt{..]} \\
 & | & \texttt{[}<\mathit{expr}>\texttt{,}<\mathit{expr}>\texttt{..}<\mathit{expr}>\texttt{]} \\
 & | & <\mathit{expr}><\mathit{op}><\mathit{expr}> \\
 & | & <\mathit{expr}><\mathit{expr}> \\
 & | & (<\mathit{expr}>) \\
 & | & \texttt{if}\ <\mathit{expr}>\ \texttt{then}\ <\mathit{expr}> \\
 & & (\texttt{elsif}\ <\mathit{expr}>\ \texttt{then}\ <\mathit{expr}>)* \\
 & & \texttt{else}\ <\mathit{expr}>\ \texttt{endif} \\
 & | & <\mathit{expr}>\ \texttt{where}\ <\mathit{definition}>+\ \texttt{endwhere} \\
 & | & \texttt{let}\ <\mathit{definition}>+\ \texttt{in}\ <\mathit{expr}>\ \texttt{endlet} \\
 & | & \backslash\ <\mathit{identifier}><\mathit{expr}>
\end{array}
$$

where \ `x E` represents the $\lambda$-expression $\lambda x.E$ .

# B The `Node` Class

The `Node` class has the following skeleton:

```
public class Node {
  public Object contents; // The object which the Node points to.

  public Node() {
    // construct a Node containing a hole (to be filled in later)
  }

  public Node(int i) {
```

```
    // construct a Node containing the long equivalent of the integer i
  }

  public Node(long l) {
    // construct a Node containing the long integer l
  }

  // ... and so on

  public Node(Object a) {
    // construct a Node containing the object a
  }

  public Node(Node a, Node f) {
    // construct a Node containing the application (f a)
  }

  public void update(Node n) {
    // update the contents of a Node so that they reflect the contents of n
  }

  public Node eval() {
    // evaluate the node to WHNF
  }

  public void print() {
    // print the contents of the node
  }
}
```

# C Code for the Unwind Method

The Java code for the unwind method of the `Node` class is as follows:

```
private Node unwind (Stack s) {
  Node vertebrae = this;
  Node whnf = null;
  do {
    // unwind to base of spine
    while (vertebrae.contents instanceof App) {
      vertebrae = ((App) vertebrae.contents).functor;
      s.push(vertebrae);
    }

    // if the base is a function see if we can apply it
    if (vertebrae.contents instanceof Method) {
      Method f = (Method) vertebrae.contents;
      int arity = f.getParameterTypes().length;

      if (s.size() - 1 < arity)
        // we have a function without enough arguments -- our original
        // node must have been in WHNF
        whnf = this;
      else {
        // Apply the function, pop its arguments and overwrite the root of
        // its redex which will then be  the top element of the stack
        Node[] args = new Node[arity];
        Node root;
```

```
        if (arity == 0) {
          // no arguments need to be popped, we just need to get the
          // root of the redex, which is left on top of the stack
          root = (Node) s.peek();
        }
        else {
          // discard the top element of the stack (another reference to f)
          s.pop();

          // get the arguments from 0 to arity - 2 (the final argument is
          // stored with the root of the redex and dealt with next)
          for (int i = 0; i < arity - 1; i++)
            args[i] = ((App)((Node) s.pop()).contents).argument;

          // get the root of the redex, which is left on top of the stack
          root = (Node) s.peek();

          // get the final argument
          args[arity - 1] = ((App)root.contents).argument;
        }

        // apply the function and update the root of the redex, a copy of
        // which is still on the top of the stack
        root.update((Node) f.invoke(null, args));

        // set our current position on the spine
        vertebrae = (Node) s.peek();
      }
    }
    else
      // else we have a primitive which cannot be reduced and is thus in WHNF
      whnf = vertebrae;
  } while (whnf == null);

  return whnf;
}
```

# D Example Programs

We detail here three of the example programs used for comparison purposes. Of the others, the four sorts are standard functions, and the `primes` and the `edigits` are discussed in Section 6

## D.1 Thue

This produces a non-repeating list of 0s, 1s and 2s.

```
thue n =
    if n <= 0 then thue1
    else take (twon n - 1) (f thue1)
    endif
        where
        twon n = if n == 0 then 1 else 2 * twon (n - 1) endif;
        sub x =
            if hd x == 0 then 0 : 1 : sub (tl x)
            else 1 : 0 : sub (tl x) endif;

        thue1 = sub (0 : tl thue1);
```

```
        f x =
            let h = hd x; t = tl x; in
                if h == 0 & hd t == 0 then
                    0 : f t
                elsif h == 0 then
                    1 : f t
                elsif hd t == 0 then
                    2 : f t
                else
                    0 : f t
                endif
            endlet;
        endwhere;

main = thue 10;
```

### D.2 Hamming

Produces the list of hamming numbers (see [3] for more details).

```
ham = 1 : foldr1 mrge (mult 2 ham : mult 3 ham : [mult 5 ham])
    where
    mult n x = map ((*) n) x;
    mrge x y =
        if isnil x then y
        elsif isnil y then x
        elsif hd x == hd y then hd x : mrge (tl x)(tl y)
        elsif hd x < hd y then hd x : mrge (tl x) y
        else hd y : mrge x (tl y)
        endif;
    endwhere;

hamming n = if n <= 0 then ham else take n ham endif;

main = hamming 100;
```

### D.3 NFib

Counts the number of function calls needed to calculate the $n$th Fibonacci number using the naive doubly-recursive method.

```
nfib n =
    if n <= 1 then
        1
    else
        1 + nfib (n - 1) + nfib (n - 2)
    endif;

main = nfib 20;
```

## References


[1] Aho, A. V., Hopcroft, J. E., and Ullman, J. D. *Data Structures and Algorithms.* Addison-Wesley, 1983.

[2] Axford, T., and Joy, M. List processing primitives for parallel computation. *Computer Languages 19*, 1 (1993), 1–17.

[3] Bird, R., and Wadler, P. *An Introduction to Functional Programming.* Prentice Hall, 1988.

[4] CORNELL, G., AND HORSTMANN, C. S. *Core Java.* Sunsoft Press (Prentice Hall), 1996.

[5] JOY, M. Ginger — a simple functional language. Tech. Rep. CS-RR-235, Department of Computer Science, Warwick University, 1992.

[6] MEEHAN, G. Fuzzy functional programming. Tech. Rep. CS-RR-322, Department of Computer Science, Warwick University, 1997.

[7] MEYER, J., AND DOWNING, T. *Java Virtual Machine.* O'Reilly, 1997.

[8] PEYTON JONES, S. L. *The Implementation of Functional Programming Languages.* Prentice Hall, 1987.

[9] PEYTON JONES, S. L., AND LESTER, D. *Implementing Functional Languages — A Tutorial.* Prentice Hall, 1992.

[10] SUN MICROSYSTEMS, INC. Java platform 1.1.2 core API. http:// java.sun.com:80/docs/.

[11] ZADEH, L. A. Outline of a new approach to the analysis of complex systems and decision processes. *IEEE Transactions on Systems, Men and Cybernetics 3* (1973), 28–44.