Virtual Machine Generator

January 7, 2013

Software Version	Date
0.0.0.1	01/01/13

Contents

1 Vmgen documentation

This documentation is for Vmgen (version 0.7.9-20120209, November 17, 2011), the virtual machine interpreter generator.

1.1 Introduction

Vmgen is a tool for writing efficient interpreters. It takes a simple virtual machine description and generates efficient C code for dealing with the virtual machine code in various ways (in particular, executing it). The run-time efficiency of the resulting interpreters is usually within a factor of 10 of machine code produced by an optimizing compiler.

The interpreter design strategy supported by Vmgen is to divide the interpreter into two parts:

- The front end takes the source code of the language to be implemented, and translates it into virtual machine code. This is similar to an ordinary compiler front end; typically an interpreter front-end performs no optimization, so it is relatively simple to implement and runs fast.
- The virtual machine interpreter executes the virtual machine code.

Such a division is usually used in interpreters, for modularity as well as for efficiency. The virtual machine code is typically passed between front end and virtual machine interpreter in memory, like in a load-and-go compiler; this avoids the complexity and time cost of writing the code to a file and reading it again.

A virtual machine (VM) represents the program as a sequence of VM instructions, following each other in memory, similar to real machine code. Control flow occurs through VM branch instructions, like in a real machine.

In this setup, Vmgen can generate most of the code dealing with virtual machine instructions from a simple description of the virtual machine instructions (see section 1.6), in particular:

VM instruction execution

VM code generation Useful in the front end.

VM code decompiler Useful for debugging the front end.

VM code tracing Useful for debugging the front end and the VM interpreter. You will typically provide other means for debugging the user's programs at the source level.

VM code profiling Useful for optimizing the VM interpreter with superinstructions (see 1.8.6).

To create parts of the interpretive system that do not deal with VM instructions, you have to use other tools (e.g., bison) and/or hand-code them.

Vmgen supports efficient interpreters though various optimizations, in particular

- Threaded code
- Caching the top-of-stack in a register
- Combining VM instructions into superinstructions
- Replicating VM (super)instructions for better BTB prediction accuracy (not yet in vmgen-ex, but already in Gforth).

As a result, Vmgen-based interpreters are only about an order of magnitude slower than native code from an optimizing C compiler on small benchmarks; on large benchmarks, which spend more time in the run-time system, the slowdown is often less (e.g., the slowdown of a Vmgen-generated JVM interpreter over the best JVM JIT compiler we measured is only a factor of 2-3 for large benchmarks; some other JITs and all other interpreters we looked at were slower than our interpreter).

VMs are usually designed as stack machines (passing data between VM instructions on a stack), and Vmgen supports such designs especially well; however, you can also use Vmgen for implementing a register VM (see Register Machines) and still benefit from most of the advantages offered by Vmgen.

There are many potential uses of the instruction descriptions that are not implemented at the moment, but we are open for feature requests, and we will consider new features if someone asks for them; so the feature list above is not exhaustive.

1.2 Why interpreters?

Interpreters are a popular language implementation technique because they combine all three of the following advantages:

- Ease of implementation
- Portability
- Fast edit-compile-run cycle

Vmgen makes it even easier to implement interpreters.

The main disadvantage of interpreters is their run-time speed. However, there are huge differences between different interpreters in this area: the slowdown over optimized C code on programs consisting of simple operations is typically a factor of 10 for the more efficient interpreters, and a factor of 1000 for the less efficient ones (the slowdown for programs executing complex operations is less, because the time spent in libraries for executing complex operations is the same in all implementation strategies).

Vmgen supports techniques for building efficient interpreters.

1.3 Concepts

- Front end and VM interpreter: Modularizing an interpretive system
- Data handling: Stacks, registers, immediate arguments
- Dispatch: From one VM instruction to the next

1.3.1 Front end and VM interpreter

Interpretive systems are typically divided into a front end that parses the input language and produces an intermediate representation for the program, and an interpreter that executes the intermediate representation of the program.

For efficient interpreters the intermediate representation of choice is virtual machine code (rather than, e.g., an abstract syntax tree). Virtual machine (VM) code consists of VM instructions arranged sequentially in memory; they are executed in sequence by the VM interpreter, but VM branch instructions can change the control flow and are used for implementing control structures. The conceptual similarity to real machine code results in the name virtual machine. Various terms similar to terms for real machines are used; e.g., there are VM registers (like the instruction pointer and stack pointer(s)), and the VM instruction consists of an opcode and immediate arguments.

In this framework, Vmgen supports building the VM interpreter and any other component dealing with VM instructions. It does not have any support for the front end, apart from VM code generation support. The front end can be implemented with classical compiler front-end techniques, supported by tools like flex and bison.

The intermediate representation is usually just internal to the interpreter, but some systems also support saving it to a file, either as an image file, or in a full-blown linkable file format (e.g., JVM). Vmgen currently has no special support for such features, but the information in the instruction descriptions can be helpful, and we are open to feature requests and suggestions.

1.3.2 Data handling

Most VMs use one or more stacks for passing temporary data between VM instructions. Another option is to use a register machine architecture for the virtual machine; we believe that using a stack architecture is usually both simpler and faster.

However, this option is slower or significantly more complex to implement than a stack machine architecture.

Vmgen has special support and optimizations for stack VMs, making their implementation easy and efficient.

You can also implement a register VM with Vmgen (see Register Machines), and you will still profit from most Vmgen features.

Stack items all have the same size, so they typically will be as wide as an integer, pointer, or floating-point value. Vmgen supports treating two consecutive stack items as a single value, but anything larger is best kept in some other memory area (e.g., the heap), with pointers to the data on the stack.

Another source of data is immediate arguments VM instructions (in the VM instruction stream). The VM instruction stream is handled similar to a stack in Vmgen.

Vmgen has no built-in support for, nor restrictions against garbage collection. If you need garbage collection, you need to provide it in your run-time libraries. Using reference counting is probably harder, but might be possible (contact us if you are interested).

1.3.3 Dispatch

Understanding this section is probably not necessary for using Vmgen, but it may help. You may want to skip it now, and read it if you find statements about dispatch methods confusing.

After executing one VM instruction, the VM interpreter has to dispatch the next VM instruction (Vmgen calls the dispatch routine '`NEXT`'). Vmgen supports two methods of dispatch:

switch dispatch In this method the VM interpreter contains a giant switch statement, with one case for each VM instruction. The VM instruction opcodes are represented by integers (e.g., produced by an `enum`) in the VM code, and dispatch occurs by loading the next opcode, switching on it, and continuing at the appropriate `case`; after executing the VM instruction, the VM interpreter jumps back to the dispatch code.

threaded code This method represents a VM instruction opcode by the address of the start of the machine code fragment for executing the VM instruction. Dispatch consists of loading this address, jumping to it, and incrementing the VM instruction pointer. Typically the threaded-code dispatch code is appended directly to the code for executing the VM instruction. Threaded code cannot be implemented in ANSI C, but it can be implemented using GNU C's labels-as-values extension (see Labels as Values).

Threaded code can be twice as fast as switch dispatch, depending on the interpreter, the benchmark, and the machine.

1.4 Invoking Vmgen

The usual way to invoke Vmgen is as follows:

```
vmgen inputfile
```

Here `inputfile` is the VM instruction description file, which usually ends in `.vmg`. The output filenames are made by taking the basename of `inputfile` (i.e., the output files will be created in the current working directory) and replacing `.vmg`

with **-vm.i**, **-disasm.i**, **-gen.i**, **-labels.i**, **-profile.i**, and **-peephole.i**. E.g., **vmgen hack/foo.vmg** will create **foo-vm.i**, **foo-disasm.i**, **foo-gen.i**, **foo-labels.i**, **foo-profile.i** and **foo-peephole.i**.

The command-line options supported by Vmgen are

--help

-h Print a message about the command-line options

--version

-v Print version and exit

1.5 Example

- Example overview
- Using profiling to create superinstructions

1.5.1 Example overview

There are two versions of the same example for using Vmgen: **vmgen-ex** and **vmgen-ex2** (you can also see Gforth as example, but it uses additional (undocumented) features, and also differs in some other respects). The example implements **mini**, a tiny Modula-2-like language with a small JavaVM-like virtual machine.

The difference between the examples is that **vmgen-ex** uses many casts, and **vmgen-ex2** tries to avoids most casts and uses unions instead. In the rest of this manual we usually mention just files in **vmgen-ex**; if you want to use unions, use the equivalent file in **vmgen-ex2**.

You can build the example by cding into the example's directory, and then typing **make**; you can check that it works with **make check**. You can run run **mini** programs like this:

```
./mini fib.mini
```

To learn about the options, type **./mini -h**.

1.5.2 Using profiling to create superinstructions

I have not added rules for this in the **Makefile** (there are many options for selecting superinstructions, and I did not want to hardcode one into the **Makefile**), but there are some supporting scripts, and here's an example:

Suppose you want to use **fib.mini** and **test.mini** as training programs, you get the profiles like this:

```
make fib.prof test.prof
```

It takes a few seconds and then you can aggregate these profiles with **stat.awk**:

```
awk -f stat.awk fib.prof test.prof
```

The result contains lines like:

```
2 16 36910041 loadlocal lit
```

This means that the sequence **`loadlocal lit`** statically occurs a total of 16 times in 2 profiles, with a dynamic execution count of 36910041.

The numbers can be used in various ways to select superinstructions. E.g., if you just want to select all sequences with a dynamic execution count exceeding 10000, you would use the following pipeline:

```
awk -f stat.awk fib.prof test.prof|
awk '$3>=10000'| #select sequences
fgrep -v -f peephole-blacklist| #eliminate wrong instructions
awk -f seq2rule.awk| #transform sequences into superinstruction rules
sort -k 3 >mini-super.vmg #sort sequences
```

The file **`peephole-blacklist`** contains all instructions that directly access a stack or stack pointer (for mini: **`call`**, **`return`**); the sort step is necessary to ensure that prefixes precede larger superinstructions.

Now you can create a version of mini with superinstructions by just saying '**`make`**'

1.6 Input File Format

Vmgen takes as input a file containing specifications of virtual machine instructions. This file usually has a name ending in **`.vmg`**.

Most examples are taken from the example in **`vmgen-ex`**.

- Input File Grammar
- Simple instructions
- Superinstructions
- Store Optimization
- Register Machines: How to define register VM instructions

1.6.1 Input File Grammar

The grammar is in EBNF format, with **`a|b`** meaning "a or b", **`{c}`** meaning 0 or more repetitions of c and **`[d]`** meaning 0 or 1 repetitions of d.

Vmgen input is not free-format, so you have to take care where you put newlines (and, in a few cases, white space).

description: {instruction|comment|eval-escape|c-escape}

instruction: simple-inst|super-inst

simple-inst: ident '(' stack-effect ')' newline c-code newline newline

stack-effect: {ident} '--' {ident}

super-inst: ident '=' ident {ident}

comment: '\ ' text newline

eval-escape: '\E ' text newline

c-escape: '\C ' text newline

Note that the \s in this grammar are meant literally, not as C-style encodings for non-printable characters.

There are two ways to delimit the C code in simple-inst:

- If you start it with a '**{**' at the start of a line (i.e., not even white space before it), you have to end it with a '**}**' at the start of a line (followed by a newline). In this case you may have empty lines within the C code (typically used between variable definitions and statements).
- You do not start it with '**{**'. Then the C code ends at the first empty line, so you cannot have empty lines within this code.

The text in **comment**, **eval-escape** and **c-escape** must not contain a newline. **Ident** must conform to the usual conventions of C identifiers (otherwise the C compiler would choke on the Vmgen output), except that idents in **stack-effect** may have a stack prefix (for stack prefix syntax, see Eval escapes).

The **c-escape** passes the text through to each output file (without the '**\C**'). This is useful mainly for conditional compilation (i.e., you write '**\C #if ...**' etc.).

In addition to the syntax given in the grammer, Vmgen also processes sync lines (lines starting with '**#line**'), as produced by '**m4 -s**' (see Invoking m4) and similar tools. This allows associating C compiler error messages with the original source of the C code.

Vmgen understands a few extensions beyond the grammar given here, but these extensions are only useful for building Gforth. You can find a description of the format used for Gforth in **prim**.

Eval escapes The text in **eval-escape** is Forth code that is evaluated when Vmgen reads the line. You will normally use this feature to define stacks and types.

If you do not know (and do not want to learn) Forth, you can build the text according to the following grammar; these rules are normally all Forth you need for using Vmgen:

```
   text:  stack-decl|type-prefix-decl|stack-prefix-decl|set-flag
   stack-decl:  'stack ' ident ident ident
   type-prefix-decl:  's" ' string '" ' ('single'|'double') ident 'type-prefix'
ident
   stack-prefix-decl:  ident 'stack-prefix' string
   set-flag:  ('store-optimization'|'include-skipped-insts') ('on'|'off')
```

Note that the syntax of this code is not checked thoroughly (there are many other Forth program fragments that could be written in an **eval-escape**).

A stack prefix can contain letters, digits, or '**:**', and may start with an '**#**'; e.g., in Gforth the return stack has the stack prefix '**R:**'. This restriction is not checked during the stack prefix definition, but it is enforced by the parsing rules for stack items later.

If you know Forth, the stack effects of the non-standard words involved are:

```
stack ( "name" "pointer" "type" -- ) ( name execution:  -- stack )
type-prefix ( addr u item-size stack "prefix" -- )
single ( -- item-size )
double ( -- item-size )
stack-prefix ( stack "prefix" -- )
store-optimization ( -- addr )
include-skipped-insts ( -- addr )
```

An **item-size** takes three cells on the stack.

1.6.2 Simple instructions

We will use the following simple VM instruction descriptions as examples:

10 ⟨*simple-instructions* 10⟩≡ (47a)

```
\ simple VM instructions:
add ( i1 i2 -- i )
i = i1+i2;

sub ( i1 i2 -- i )
i = i1-i2;

mul ( i1 i2 -- i )
i = i1*i2;

and ( i1 i2 -- i )
i = i1 & i2;

or ( i1 i2 -- i )
i = i1 | i2;

lessthan ( i1 i2 -- i )
i = i1<i2;

equals ( i1 i2 -- i )
i = i1==i2;

not ( i1 -- i2 )
i2 = !i1;

negate ( i1 -- i2 )
i2 = -i1;
```

⟨*lit* 11c⟩

```
drop ( i -- )
```

```
print ( i -- )
printf("%ld\n", i);
```

The first line specifies the name of the VM instruction (e.g. **sub**) and its stack effect (e.g. **i1 i2 -- i**). The rest of the description is just plain C code.

The stack effect specifies that sub pulls two integers from the data stack and puts them in the C variables **i1** and **i2** (with the rightmost item (**i2**) taken from the top of stack; intuition: if you push **i1**, then **i2** on the stack, the resulting stack picture is **i1 i2**) and later pushes one integer (**i**) on the data stack (the rightmost item is on the top afterwards).

How do we know the type and stack of the stack items? Vmgen uses prefixes, similar to Fortran; in contrast to Fortran, you have to define the prefix first:

11a ⟨*type-prefix* 11a⟩≡ (47a)

```
\ type prefix definitions:
\E s" Cell"   single data-stack type-prefix i
\E s" char *" single data-stack type-prefix a
\E s" Inst *" single data-stack type-prefix target
```

This defines the prefix **i** to refer to the type **Cell** (defined as **long** in **mini.h**) and, by default, to the **data-stack**. It also specifies that this type takes one stack item (**single**). The type prefix is part of the variable name.

Before we can use **data-stack** in this way, we have to define it:

11b ⟨*stack-definitions* 11b⟩≡ (47a)

```
\ stack definitions:
\E stack data-stack sp Cell
```

This line defines the stack data-stack, which uses the stack pointer **sp**, and each item has the basic type **Cell**; other types have to fit into one or two Cells (depending on whether the type is single or double wide), and are cast from and to Cells on accessing the data-stack with type cast macros (see VM engine). By default, stacks grow towards lower addresses in Vmgen-erated interpreters (see Stack growth direction).

We can override the default stack of a stack item by using a stack prefix. E.g., consider the following instruction:

11c ⟨*lit* 11c⟩≡ (10)

```
lit ( #i -- i )
```

The VM instruction **lit** takes the item **i** from the instruction stream (indicated by the prefix **#**), and pushes it on the (default) data stack. The stack prefix is not part of the variable name. Stack prefixes are defined like this:

11d ⟨*stack-prefix* 11d⟩≡ (47a)

```
\ stack prefix definitions
\E inst-stream stack-prefix #
\E data-stack  stack-prefix S:
```

This definition defines that the stack prefix **#** specifies the "stack" **inst-stream**. Since the instruction stream behaves a little differently than an ordinary stack, it is predefined, and you do not need to define it.

The instruction stream contains instructions and their immediate arguments, so specifying that an argument comes from the instruction stream indicates an immediate argument. Of course, instruction stream arguments can only appear to the left of **--** in the stack effect. If there are multiple instruction stream arguments, the leftmost is the first one (just as the intuition suggests).

Explicit stack access This feature is not needed and not supported in the 0.7.9 version of vmgen that is documented here (and that is invoked by default).

Not all stack effects can be specified using the stack effect specifications above. For VM instructions that have other stack effects, you can specify them explicitly by accessing the stack pointer in the C code; however, you have to notify Vmgen of such explicit stack accesses, otherwise Vmgens optimizations could conflict with your explicit stack accesses.

You notify Vmgen by putting **...** with the appropriate stack prefix into the stack comment. Then the VM instruction will first take the other stack items specified in the stack effect into C variables, then make sure that all other stack items for that stack are in memory, and that the stack pointer for the stack points to the top-of-stack (by default, unless you change the stack access transformation: see Stack growth direction).

The general rule is: If you mention a stack pointer in the C code of a VM instruction, you should put a **...** for that stack in the stack effect.

Consider this example:

12a ⟨*unused-stack-adjust* 12a⟩≡
```
return ( #iadjust S:... target afp i1 -- i2 )
SET_IP(target);
sp = (Cell *)(((char *)sp)+iadjust);
fp = afp;
i2=i1;
```

First the variables **target afp i1** are popped off the stack, then the stack pointer **sp** is set correctly for the new stack depth, then the C code changes the stack depth and does other things, and finally **i2** is pushed on the stack with the new depth.

The position of the **...** within the stack effect does not matter. You can use several **...**s, for different stacks, and also several for the same stack (that has no additional effect). If you use **...** without a stack prefix, this specifies all the stacks except the instruction stream.

You cannot use **...** for the instruction stream, but that is not necessary: At the start of the C code, IP points to the start of the next VM instruction (i.e., right beyond the end of the current VM instruction), and you can change the instruction pointer with **SET_IP** (see VM engine).

C Code Macros Vmgen recognizes the following strings in the C code part of simple instructions:

SET_IP As far as Vmgen is concerned, a VM instruction containing this ends a VM basic block (used in profiling to delimit profiled sequences). On the C level, this also sets the instruction pointer.

12b ⟨*branch* 12b⟩≡ (47a)
```
branch ( #target -- )
SET_IP(target);
```

SUPER_END This ends a basic block (for profiling), even if the instruction contains no **SET_IP**.

13a ⟨*end* 13a⟩≡ (47a)
```
end ( i -- )
/* SUPER_END would increment the next BB count (because IP points there);
   this would be a problem if there is no following BB.
   Instead, we do the following to add an end point for the current BB: */
#ifdef VM_PROFILING
block_insert(IP); /* we also do this at compile time, so this is unnecessary */
#endif
return i;
```

INST_TAIL; Vmgen replaces '**INST_TAIL;**' with code for ending a VM instruction and dispatching the next VM instruction. Even without a '**INST_TAIL;**' this happens automatically when control reaches the end of the C code. If you want to have this in the middle of the C code, you need to use '**INST_TAIL;**'. A typical example is a conditional VM branch. In this example, '**INST_TAIL;**' is not strictly necessary, because there is another one implicitly after the if-statement, but using it improves branch prediction accuracy slightly and allows other optimizations.

13b ⟨*zbranch* 13b⟩≡ (47a)
```
zbranch ( #target i -- )
if (i==0) {
  SET_IP(target);
  INST_TAIL;
}
```

SUPER_CONTINUE This indicates that the implicit tail at the end of the VM instruction dispatches the sequentially next VM instruction even if there is a **SET_IP** in the VM instruction. This enables an optimization that is not yet implemented in the vmgen-ex code (but in Gforth). The typical application is in conditional VM branches.

13c ⟨*unused-branch* 13c⟩≡
```
unused-branch ( #target i -- )
if (i==0) {
  SET_IP(target);
  INST_TAIL;
}
SUPER_CONTINUE;
```

Note that Vmgen is not smart about C-level tokenization, comments, strings, or conditional compilation, so it will interpret even a commented-out **SUPER_END** as ending a basic block (or, e.g., '**RESET_IP**;' as '**SET_IP**;'). Conversely, Vmgen requires the literal presence of these strings; Vmgen will not see them if they are hiding in a C preprocessor macro.

C Code restrictions Vmgen generates code and performs some optimizations under the assumption that the user-supplied C code does not access the stack pointers or stack items, and that accesses to the instruction pointer only occur through special macros. In general you should heed these restrictions. However, if you need to break these restrictions, read the following.

Accessing a stack or stack pointer directly can be a problem for several reasons:

- Vmgen optionally supports caching the top-of-stack item in a local variable (that is allocated to a register). This is the most frequent source of trouble. You can deal with it either by not using top-of-stack caching (slowdown factor 1-1.4, depending on machine), or by inserting flushing code (e.g., '**`IF_spTOS(sp[...] = spTOS);`**') at the start and reloading code (e.g., '**`IF_spTOS(spTOS = sp[0])`**') at the end of problematic C code. Vmgen inserts a stack pointer update before the start of the user-supplied C code, so the flushing code has to use an index that corrects for that. In the future, this flushing may be done automatically by mentioning a special string in the C code.

14 ⟨*stack-caching* 14⟩≡ (47a)

```
\ The following VM instructions also explicitly reference sp and
\ therefore may have to do something about spTOS caching.

call ( #target #iadjust -- targetret aoldfp )
/* IF_spTOS(sp[2] = spTOS);*/ /* unnecessary; vmgen inserts a flush anyway */
targetret = IP;
SET_IP(target);
aoldfp = fp;
sp = (Cell *)(((char *)sp)+iadjust);
fp = (char *)sp;
/* IF_spTOS(spTOS = sp[0]); */ /* dead, thus unnecessary; vmgen copies aoldfp there */

return ( #iadjust target afp i1 -- i2 )
/* IF_spTOS(sp[-2] = spTOS); */ /* unnecessary; that stack item is dead */
SET_IP(target);
sp = (Cell *)(((char *)sp)+iadjust);
fp = afp;
i2=i1;
/* IF_spTOS(spTOS = sp[0]); */ /* dead, thus unnecessary; vmgen copies i2 there */

\ loadlocal and storelocal access stack items below spTOS, so we can
\ ignore spTOS caching.

loadlocal ( #ioffset -- i )
i = *(Cell *)(fp+ioffset);

storelocal ( #ioffset i -- )
*(Cell *)(fp+ioffset) = i;
```

- The Vmgen-erated code loads the stack items from stack-pointer-indexed memory into variables before the user-supplied C code, and stores them from variables to stack-pointer-indexed memory afterwards. If you do any writes to the stack through its stack pointer in your C code, it will not affect the variables, and your write may be overwritten by the stores after the C code. Similarly, a read from a stack using a stack pointer will not reflect computations of stack items in the same VM instruction.
- Superinstructions keep stack items in variables across the whole superinstruction. So you should not include VM instructions, that access a stack or stack pointer, as components of superinstructions (see VM profiler).

You should access the instruction pointer only through its special macros ('**IP**', '**SET_IP**', '**IPTOS**'); this ensure that these macros can be implemented in several ways for best performance. '**IP**' points to the next instruction, and '**IPTOS**' is its contents.

Stack growth direction By default, the stacks grow towards lower addresses. You can change this for a stack by setting the **stack-access-transform** field of the stack to an xt (**itemnum -- index**) that performs the appropriate index transformation.

E.g., if you want to let **data-stack** grow towards higher addresses, with the stack pointer always pointing just beyond the top-of-stack, use this right after defining **data-stack**:

15a ⟨*unused-stack-transform* 15a⟩≡
```
\E : sp-access-transform ( itemnum -- index ) negate 1- ;
\E ' sp-access-transform ' data-stack >body stack-access-transform !
```

This means that **sp-access-transform** will be used to generate indexes for accessing **data-stack**. The definition of **sp-access-transform** above transforms **n** into **-n-1**, e.g, 1 into -2. This will access the 0th data-stack element (top-of-stack) at sp[-1], the 1st at sp[-2], etc., which is the typical way upward-growing stacks are used. If you need a different transform and do not know enough Forth to program it, let me know.

15b ⟨*stack-organization* 15b⟩≡ (47a)
```
\  The stack is organized as follows:
\  The stack grows downwards; a stack usually looks like this:

\  higher addresses
\  --------------------- bottom of stack
\     locals of main
\     return address (points to VM code after call)
\  +->oldfp (NULL)
\  |  intermediate results (e.g., 1 for a call like 1+foo(...))
\  |  arguments passed to the called function
\  |  locals of the called function
\  |  return address (points to VM code after call)
\  +--oldfp               <-- fp
\     intermediate results <-- sp
\  ---------------------- top of stack
\  lower addresses
```

1.6.3 Superinstructions

Note: don't invest too much work in (static) superinstructions; a future version of Vmgen will support dynamic superinstructions (see Ian Piumarta and Fabio Riccardi, Optimizing Direct Threaded Code by Selective Inlining, PLDI'98), and static superinstructions have much less benefit in that context (preliminary results indicate only a factor 1.1 speedup).

Here is an example of a superinstruction definition:

16 ⟨*ll* 16⟩≡ (47b)

```
ll = loadlocal lit
```

ll is the name of the superinstruction, and **loadlocal** and **lit** are its components. This superinstruction performs the same action as the sequence **loadlocal** and **lit**. It is generated automatically by the VM code generation functions whenever that sequence occurs, so if you want to use this superinstruction, you just need to add this definition (and even that can be partially automatized, see VM profiler).

Vmgen requires that the component instructions are simple instructions defined before superinstructions using the components. Currently, Vmgen also requires that all the subsequences at the start of a superinstruction (prefixes) must be defined as superinstruction before the superinstruction. I.e., if you want to define a superinstruction

```
foo4 = load add sub mul
```

you first have to define **load**, **add**, **sub** and **mul**, plus

```
foo2 = load add
foo3 = load add sub
```

Here, **foo2** is the longest prefix of **foo3**, and **foo3** is the longest prefix of **foo4**.

Note that Vmgen assumes that only the code it generates accesses stack pointers, the instruction pointer, and various stack items, and it performs optimizations based on this assumption. Therefore, VM instructions where your C code changes the instruction pointer should only be used as last component; a VM instruction where your C code accesses a stack pointer should not be used as component at all. Vmgen does not check these restrictions, they just result in bugs in your interpreter.

The Vmgen flag **include-skipped-insts** influences superinstruction code generation. Currently there is no support in the peephole optimizer for both variations, so leave this flag alone for now.

1.6.4 Store Optimization

This minor optimization (0.6%–0.8% reduction in executed instructions for Gforth) puts additional requirements on the instruction descriptions and is therefore disabled by default.

What does it do? Consider an instruction like

```
dup ( n -- n n )
```

For simplicity, also assume that we are not caching the top-of-stack in a register. Now, the C code for dup first loads n from the stack, and then stores it twice to the stack, one time to the address where it came from; that time is unnecessary, but gcc does not optimize it away, so vmgen can do it instead (if you turn on the store optimization).

Vmgen uses the stack item's name to determine if the stack item contains the same value as it did at the start. Therefore, if you use the store optimization, you have to ensure that stack items that have the same name on input and output also have the same value, and are not changed in the C code you supply. I.e., the following code could fail if you turn on the store optimization:

```
add1 ( n -- n )
n++;
```

Instead, you have to use different names, i.e.:

```
add1 ( n1 -- n2 )
n2=n1+1;
```

Similarly, the store optimization assumes that the stack pointer is only changed by Vmgen-erated code. If your C code changes the stack pointer, use different names in input and output stack items to avoid a (probably wrong) store optimization, or turn the store optimization off for this VM instruction.

To turn on the store optimization, write

```
\E store-optimization on
```

at the start of the file. You can turn this optimization on or off between any two VM instruction descriptions. For turning it off again, you can use

```
\E store-optimization off
```

1.6.5 Register Machines

If you want to implement a register VM rather than a stack VM with Vmgen, there are two ways to do it: Directly and through superinstructions.

If you use the direct way, you define instructions that take the register numbers as immediate arguments, like this:

```
add3 ( #src1 #src2 #dest -- )
reg[dest] = reg[src1]+reg[src2];
```

A disadvantage of this method is that during tracing you only see the register numbers, but not the register contents. Actually, with an appropriate definition of **`printarg_src`** (see VM engine), you can print the values of the source registers on entry, but you cannot print the value of the destination register on exit.

If you use superinstructions to define a register VM, you define simple instructions that use a stack, and then define superinstructions that have no overall stack effect, like this:

```
loadreg ( #src -- n )
n = reg[src];
storereg ( n #dest -- )
reg[dest] = n;
adds ( n1 n2 -- n )
n = n1+n2;
add3 = loadreg loadreg adds storereg
```

An advantage of this method is that you see the values and not just the register numbers in tracing. A disadvantage of this method is that currently you cannot generate superinstructions directly, but only through generating a sequence of simple instructions (we might change this in the future if there is demand).

Could the register VM support be improved, apart from the issues mentioned above? It is hard to see how to do it in a general way, because there are a number of different designs that different people mean when they use the term register machine in connection with VM interpreters. However, if you have ideas or requests in that direction, please let me know (see Contact).

1.7 Error messages

These error messages are created by Vmgen:

`# can only be on the input side`

You have used an instruction-stream prefix (usually '#') after the '--' (the output side); you can only use it before (the input side).

`the prefix for this superinstruction must be defined earlier`

You have defined a superinstruction (e.g. abc = a b c) without defining its direct prefix (e.g., ab = a b), See Superinstructions.

`sync line syntax`

If you are using a preprocessor (e.g., m4) to generate Vmgen input code, you may want to create #line directives (aka sync lines). This error indicates that such a line is not in th syntax expected by Vmgen (this should not happen; please report the offending line in a bug report).

`syntax error, wrong char`

A syntax error. If you do not see right away where the error is, it may be helpful to check the following: Did you put an empty line in a VM instruction where the C code is not delimited by braces (then the empty line ends the VM instruction)? If you used brace-delimited C code, did you put the delimiting braces (and only those) at the start of the line, without preceding white space? Did you forget a delimiting brace?

`too many stacks`

Vmgen currently supports 3 stacks (plus the instruction stream); if you need more, let us know.

`unknown prefix`

The stack item does not match any defined type prefix (after stripping away any stack prefix). You should either declare the type prefix you want for that stack item, or use a different type prefix

`unknown primitive`

You have used the name of a simple VM instruction in a superinstruction definition without defining the simple VM instruction first.

In addition, the C compiler can produce errors due to code produced by Vmgen; e.g., you need to define type cast functions.

1.8 Using the generated code

The easiest way to create a working VM interpreter with Vmgen is probably to start with **`vmgen-ex`**, and modify it for your purposes. This chapter explains what the various wrapper and generated files do. It also contains reference-manual style descriptions of the macros, variables etc. used by the generated code, and you can skip that on first reading.

- VM engine: Executing VM code
- VM instruction table
- VM code generation: Creating VM code (in the front-end)
- Peephole optimization: Creating VM superinstructions
- VM disassembler: for debugging the front end
- VM profiler: for finding worthwhile superinstructions

1.8.1 VM engine

The VM engine is the VM interpreter that executes the VM code. It is essential for an interpretive system.

Vmgen supports two methods of VM instruction dispatch: threaded code (fast, but gcc-specific), and switch dispatch (slow, but portable across C compilers); you can use conditional compilation (‘`defined(__GNUC__)`’) to choose between these methods, and our example does so.

20 ⟨*dispatch* 20⟩≡ (61)

```
/* different threading schemes for different architectures; the sparse
   numbering is there for historical reasons */

/* here you select the threading scheme; I have only set this up for
   386 and generic, because I don't know what preprocessor macros to
   test for (Gforth uses config.guess instead).  Anyway, it's probably
   best to build them all and select the fastest instead of hardwiring
   a specific scheme for an architecture.  E.g., scheme 8 is fastest
   for Gforth "make bench" on a 486, whereas scheme 5 is fastest for
   "mini fib.mini" on an Athlon */
#ifndef THREADING_SCHEME
#define THREADING_SCHEME 5
#endif /* defined(THREADING_SCHEME) */

#ifdef __GNUC__
#if THREADING_SCHEME==1
/* direct threading scheme 1: autoinc, long latency (HPPA, Sharc) */
#   ⟨NEXT-P0-1 24a⟩
#   ⟨IP-1 26c⟩
#   ⟨SET-IP-1 25g⟩
#   ⟨NEXT-INST-1 24l⟩
#   ⟨INC-IP-1 25c⟩
#   ⟨DEF-CA 23d⟩
#   ⟨NEXT-P1 24c⟩
#   ⟨NEXT-P2-1 24h⟩
#endif

#if THREADING_SCHEME==3
/* direct threading scheme 3: autoinc, low latency (68K) */
#   ⟨NEXT-P0 23e⟩
#   ⟨IP-2 26d⟩
#   ⟨SET-IP-1 25g⟩
#   ⟨NEXT-INST 24k⟩
#   ⟨INC-IP-2 25d⟩
#   ⟨DEF-CA 23d⟩
#   ⟨NEXT-P1-1 24d⟩
#   ⟨NEXT-P2-1 24h⟩
#endif
```

```
#if THREADING_SCHEME==5
/* direct threading scheme 5: early fetching (Alpha, MIPS) */
#  define CFA_NEXT
#  ⟨NEXT-P0-2 24b⟩
#  ⟨IP-2 26d⟩
#  ⟨SET-IP-1 25g⟩
#  ⟨NEXT-INST-1 24l⟩
#  ⟨INC-IP-1 25c⟩
#  ⟨DEF-CA 23d⟩
#  ⟨NEXT-P1-2 24e⟩
#  ⟨NEXT-P2-1 24h⟩
#endif

#if THREADING_SCHEME==8
/* direct threading scheme 8: i386 hack */
#  ⟨NEXT-P0 23e⟩
#  ⟨IP-2 26d⟩
#  ⟨SET-IP-1 25g⟩
#  ⟨NEXT-INST 24k⟩
#  ⟨INC-IP-2 25d⟩
#  ⟨DEF-CA 23d⟩
#  ⟨NEXT-P1-2 24e⟩
#  ⟨NEXT-P2-2 24i⟩
#endif

#if THREADING_SCHEME==9
/* direct threading scheme 9: prefetching (for PowerPC) */
/* note that the "cfa=next_cfa;" occurs only in NEXT_P1, because this
   works out better with the capabilities of gcc to introduce and
   schedule the mtctr instruction. */
#  ⟨NEXT-P0 23e⟩
#  ⟨IP 26b⟩
#  ⟨SET-IP-2 25h⟩
#  ⟨NEXT-INST-2 25a⟩
#  ⟨INC-IP-3 25e⟩
#  ⟨DEF-CA 23d⟩
#  ⟨NEXT-P1-3 24f⟩
#  ⟨NEXT-P2-1 24h⟩
#  define MORE_VARS     Inst next_cfa;
#endif

#if THREADING_SCHEME==10
/* direct threading scheme 10: plain (no attempt at scheduling) */
#  ⟨NEXT-P0 23e⟩
#  ⟨IP-2 26d⟩
#  ⟨SET-IP-1 25g⟩
```

```
#   ⟨NEXT-INST 24k⟩
#   ⟨INC-IP-2 25d⟩
#   ⟨DEF-CA 23d⟩
#   ⟨NEXT-P1 24c⟩
#   ⟨NEXT-P2-3 24j⟩
#endif

#define NEXT ({DEF_CA NEXT_P1; NEXT_P2;})
#⟨IPTOS 27c⟩

#⟨jump-INST-ADDR 28b⟩
#⟨jump-LABEL 22⟩
#else /* !defined(__GNUC__) */
/* use switch dispatch */
#⟨DEF-CA 23d⟩
#⟨NEXT-P0 23e⟩
#⟨NEXT-P1 24c⟩
#⟨NEXT-P2 24g⟩
#⟨SET-IP 25f⟩
#⟨IP 26b⟩
#⟨NEXT-INST 24k⟩
#⟨INC-IP 25b⟩
#⟨IPTOS 27c⟩
#⟨switch-INST-ADDR 29⟩
#⟨switch-LABEL 23a⟩

#endif /* !defined(__GNUC__) */
```

For both methods, the VM engine is contained in a C-level function. Vmgen generates most of the contents of the function for you (name-vm.i), but you have to define this function, and macros and variables used in the engine, and initialize the variables. In our example the engine function also includes name-labels.i (see VM instruction table).

In addition to executing the code, the VM engine can optionally also print out a trace of the executed instructions, their arguments and results. For superinstructions it prints the trace as if only component instructions were executed; this allows to introduce new superinstructions while keeping the traces comparable to old ones (important for regression tests).

It costs significant performance to check in each instruction whether to print tracing code, so we recommend producing two copies of the engine: one for fast execution, and one for tracing. See the rules for engine.o and engine-debug.o in vmgen-ex/Makefile for an example.

The following macros and variables are used in name-vm.i:

LABEL(inst_name) This is used just before each VM instruction to provide a jump or switch label (the '`:`' is provided by Vmgen). For switch dispatch this should expand to '`case label:`'; for threaded-code dispatch this should just expand to '`label:`'. In either case label is usually the `inst_name` with some prefix or suffix to avoid naming conflicts.

22 ⟨*jump-LABEL* 22⟩≡ (20)

```
define LABEL(name) I_##name:
```

23a ⟨*switch-LABEL* 23a⟩≡ (20)
```
define LABEL(name) case I_##name:
```

LABEL2(inst_name) This will be used for dynamic superinstructions; at the moment, this should expand to nothing.

23b ⟨*LABEL2* 23b⟩≡ (61)
```
#define LABEL2(x)
```

NAME(inst_name_string) Called on entering a VM instruction with a string containing the name of the VM instruction as parameter. In normal execution this should be expand to nothing, but for tracing this usually prints the name, and possibly other information (several VM registers in our example).

23c ⟨*NAME* 23c⟩≡ (61)
```
#ifdef VM_DEBUG
#define NAME(_x) if (vm_debug) {fprintf(vm_out, "%p: %-20s, ", ip-1, _x); \
                                fprintf(vm_out,"fp=%p, sp=%p", fp, sp);}
#else
#define NAME(_x)
#endif
```

DEF_CA Usually empty. Called just inside a new scope at the start of a VM instruction. Can be used to define variables that should be visible during every VM instruction. If you define this macro as non-empty, you have to provide the finishing '`;`' in the macro.

23d ⟨*DEF-CA* 23d⟩≡ (20)
```
define DEF_CA
```

NEXT_P0 NEXT_P1 NEXT_P2 The three parts of instruction dispatch. They can be defined in different ways for best performance on various processors (see engine.c in the example or engine/threaded.h in Gforth). '**`NEXT_P0`**' is invoked right at the start of the VM instruction (but after '**`DEF_CA`**'), '**`NEXT_P1`**' right after the user-supplied C code, and '**`NEXT_P2`**' at the end. The actual jump has to be performed by '**`NEXT_P2`**' (if you would do it earlier, important parts of the VM instruction would not be executed). The simplest variant is if '**`NEXT_P2`**' does everything and the other macros do nothing. Then also related macros like '**`IP`**', '**`SET_IP`**', '**`INC_IP`**' and '**`IPTOS`**' are very straightforward to define. For switch dispatch this code consists just of a jump to the dispatch code ('`goto next_inst;`' in our example); for direct threaded code it consists of something like '({cfa=*ip++; goto *cfa;})'. Pulling code (usually the '`cfa=*ip++;`') up into '**`NEXT_P1`**' usually does not cause problems, but pulling things up into '**`NEXT_P0`**' usually requires changing the other macros (and, at least for Gforth on Alpha, it does not buy much, because the compiler often manages to schedule the relevant stuff up by itself). An even more extreme variant is to pull code up even further, into, e.g., **`NEXT_P1`** of the previous VM instruction (prefetching, useful on PowerPCs).

23e ⟨*NEXT-P0* 23e⟩≡ (20)
```
define NEXT_P0
```

24a ⟨*NEXT-P0-1* 24a⟩≡ (20)
```
define NEXT_P0      ({cfa=*ip++;})
```

24b ⟨*NEXT-P0-2* 24b⟩≡ (20)
```
define NEXT_P0      ({cfa=*ip;})
```

24c ⟨*NEXT-P1* 24c⟩≡ (20)
```
define NEXT_P1
```

24d ⟨*NEXT-P1-1* 24d⟩≡ (20)
```
define NEXT_P1      ({cfa=*ip++;})
```

24e ⟨*NEXT-P1-2* 24e⟩≡ (20)
```
define NEXT_P1      (ip++)
```

24f ⟨*NEXT-P1-3* 24f⟩≡ (20)
```
define NEXT_P1      ({cfa=next_cfa; ip++; next_cfa=*ip;})
```

24g ⟨*NEXT-P2* 24g⟩≡ (20)
```
define NEXT_P2 goto next_inst;
```

24h ⟨*NEXT-P2-1* 24h⟩≡ (20)
```
define NEXT_P2      ({goto *cfa;})
```

24i ⟨*NEXT-P2-2* 24i⟩≡ (20)
```
define NEXT_P2      ({goto **(ip-1);})
```

24j ⟨*NEXT-P2-3* 24j⟩≡ (20)
```
define NEXT_P2      ({cfa=*ip++; goto *cfa;})
```

24k ⟨*NEXT-INST* 24k⟩≡ (20)
```
define NEXT_INST    (*ip)
```

24l ⟨*NEXT-INST-1* 24l⟩≡ (20)
```
define NEXT_INST    (cfa)
```

25a ⟨*NEXT-INST-2* 25a⟩≡ (20)
```
define NEXT_INST     (next_cfa)
```

INC_IP(n) This increments IP by n.

25b ⟨*INC-IP* 25b⟩≡ (20)
```
define INC_IP(const_inc) (ip+=(const_inc))
```

25c ⟨*INC-IP-1* 25c⟩≡ (20)
```
define INC_IP(const_inc) ({cfa=IP[const_inc]; ip+=(const_inc);})
```

25d ⟨*INC-IP-2* 25d⟩≡ (20)
```
define INC_IP(const_inc) ({ip+=(const_inc);})
```

25e ⟨*INC-IP-3* 25e⟩≡ (20)
```
define INC_IP(const_inc) ({next_cfa=IP[const_inc]; ip+=(const_inc);})
```

SET_IP(target) This sets IP to target.

25f ⟨*SET-IP* 25f⟩≡ (20)
```
define SET_IP(p)     (ip=(p))
```

25g ⟨*SET-IP-1* 25g⟩≡ (20)
```
define SET_IP(p)     ({ip=(p); NEXT_P0;})
```

25h ⟨*SET-IP-2* 25h⟩≡ (20)
```
define SET_IP(p)     ({ip=(p); next_cfa=*ip; NEXT_P0;})
```

vm_A2B(a,b) Type casting macro that assigns ‘a’ (of type A) to ‘b’ (of type B). This is mainly used for getting stack items into variables and back. So you need to define macros for every combination of stack basic type (Cell in our example) and type-prefix types used with that stack (in both directions). For the type-prefix type, you use the type-prefix (not the C type string) as type name (e.g., ‘vm_Cell2i’, not ‘vm_Cell2Cell’). In addition, you have to define a vm_X2X macro for the stack’s basic type X (used in superinstructions). The stack basic type for the predefined ‘inst-stream’ is ‘Cell’. If you want a stack with the same item size, making its basic type ‘Cell’ usually reduces the number of macros you have to define. Here our examples differ a lot: vmgen-ex uses casts in these macros, whereas vmgen-ex2 uses union-field selection (or assignment to union fields). Note that casting floats into integers and vice versa changes the bit pattern (and you do not want that). In this case your options are to use a (temporary) union, or to take the address of the value, cast the pointer, and dereference that (not always possible, and sometimes expensive).

26a ⟨*vm-A2B(a,b)* 26a⟩≡ (45)

```
/* type change macros; these are specific to the types you use, so you
   have to change this part */
#define vm_Cell2i(_cell,x)      ((x)=(long)(_cell))
#define vm_Cell2target(_cell,x) ((x)=(Inst *)(_cell))
#define vm_Cell2a(_cell,x)      ((x)=(char *)(_cell))
#define vm_i2Cell(x,_cell)      ((_cell)=(Cell)(x))
#define vm_target2Cell(x,_cell) ((_cell)=(Cell)(x))
#define vm_a2Cell(x,_cell)      ((_cell)=(Cell)(x))
#define vm_Cell2Cell(_x,_y)     ((_y)=(Cell)(_x))
/* the cast in vm_Cell2Cell is needed because the base type for
   inst-stream is Cell, but *IP is an Inst */
```

vm_twoA2B(a1,a2,b) vm_B2twoA(b,a1,a2) Type casting between two stack items (a1, a2) and a variable b of a type that takes two stack items. This does not occur in our small examples, but you can look at Gforth for examples (see vm_twoCell2d in engine/forth.h).

stackpointer For each stack used, the stackpointer name given in the stack declaration is used. For a regular stack this must be an l-expression; typically it is a variable declared as a pointer to the stack’s basic type. For ‘inst-stream’, the name is ‘**IP**’, and it can be a plain r-value; typically it is a macro that abstracts away the differences between the various implementations of **NEXT_P***.

26b ⟨*IP* 26b⟩≡ (20)

```
define IP           ip
```

26c ⟨*IP-1* 26c⟩≡ (20)

```
define IP           (ip-1)
```

26d ⟨*IP-2* 26d⟩≡ (20)

```
define IP           (ip)
```

IMM_ARG(access,value) Define this to expland to "(access)". This is just a placeholder for future extensions.

27a ⟨*IMM-ARG* 27a⟩≡ (45)
```
/* for future extensions */
#define IMM_ARG(access,value) (access)
```

stackpointerTOS The top-of-stack for the stack pointed to by stackpointer. If you are using top-of-stack caching for that stack, this should be defined as variable; if you are not using top-of-stack caching for that stack, this should be a macro expanding to 'stackpointer[0]'. The stack pointer for the predefined 'inst-stream' is called 'IP', so the top-of-stack is called 'IPTOS'.

27b ⟨*spTOS* 27b⟩≡ (61)
```
#ifdef USE_spTOS
  Cell   spTOS;
#else
#define spTOS (sp[0])
#endif
```

27c ⟨*IPTOS* 27c⟩≡ (20)
```
define IPTOS NEXT_INST
```

IF_stackpointerTOS(expr) Macro for executing expr, if top-of-stack caching is used for the stackpointer stack. I.e., this should do expr if there is top-of-stack caching for stackpointer; otherwise it should do nothing.

27d ⟨*IF-spTOS* 27d⟩≡ (61)
```
#define USE_spTOS 1

#ifdef USE_spTOS
#define IF_spTOS(x) x
#else
#define IF_spTOS(x)
#endif
```

SUPER_END This is used by the VM profiler (see VM profiler); it should not do anything in normal operation, and call **`vm_count_block(IP)`** for profiling.

27e ⟨*SUPER-END* 27e⟩≡ (61)
```
#ifdef VM_PROFILING
#define SUPER_END  vm_count_block(IP)
#else
#define SUPER_END
#endif
```

SUPER_CONTINUE This is just a hint to Vmgen and does nothing at the C level. See 1.6.2

MAYBE_UNUSED This should be defined as **__attribute__((unused))** for gcc-2.7 and higher. It suppresses the warnings about unused variables in the code for superinstructions. You need to define this only if you are using superinstructions.

28a ⟨*MAYBE-UNUSED* 28a⟩≡ (61)
```
#if defined(__GNUC__) && ((__GNUC__==2 && defined(__GNUC_MINOR__) && __GNUC_MINOR__>=7)||(_
#define MAYBE_UNUSED __attribute__((unused))
#else
#define MAYBE_UNUSED
#endif
```

VM_DEBUG If this is defined, the tracing code will be compiled in (slower interpretation, but better debugging). Our example compiles two versions of the engine, a fast-running one that cannot trace, and one with potential tracing and profiling.

vm_debug Needed only if 'VM_DEBUG' is defined. If this variable contains true, the VM instructions produce trace output. It can be turned on or off at any time.

vm_out Needed only if 'VM_DEBUG' is defined. Specifies the file on which to print the trace output (type 'FILE *').

printarg_type(value) Needed only if 'VM_DEBUG' is defined. Macro or function for printing value in a way appropriate for the type. This is used for printing the values of stack items during tracing. Type is normally the type prefix specified in a type-prefix definition (e.g., 'printarg_i'); in superinstructions it is currently the basic type of the stack.

1.8.2 VM instruction table

For threaded code we also need to produce a table containing the labels of all VM instructions. This is needed for VM code generation (see VM code generation), and it has to be done in the engine function, because the labels are not visible outside. It then has to be passed outside the function (and assigned to '**vm_prim**'), to be used by the VM code generation functions.

This means that the engine function has to be called first to produce the VM instruction table, and later, after generating VM code, it has to be called again to execute the generated VM code (yes, this is ugly). In our example program, these two modes of calling the engine function are differentiated by the value of the parameter **ip0** (if it equals 0, then the table is passed out, otherwise the VM code is executed); in our example, we pass the table out by assigning it to '**vm_prim**' and returning from '**engine**'.

In our example (**vmgen-ex/engine.c**), we also build such a table for switch dispatch; this is mainly done for uniformity.

For switch dispatch, we also need to define the VM instruction opcodes used as case labels in an enum.

For both purposes (VM instruction table, and enum), the file **name-labels.i** is generated by Vmgen. You have to define the following macro used in this file:

INST_ADDR(inst_name) For switch dispatch, this is just the name of the switch label (the same name as used in '**LABEL(inst_name)**') for both uses of name-labels.i. For threaded-code dispatch, this is the address of the label defined in '**LABEL(inst_name)**'); the address is taken with '&&' (see Labels as Values).

28b ⟨*jump-INST-ADDR* 28b⟩≡ (20)
```
define INST_ADDR(name) (Label)&&I_##name
```

29 ⟨*switch-INST-ADDR* 29⟩≡ (20)

```
define INST_ADDR(name) I_##name
```

1.8.3 VM code generation

Vmgen generates VM code generation functions in **name-gen.i** that the front end can call to generate VM code. This is essential for an interpretive system.

For a VM instruction '**x (#a b #c -- d)**', Vmgen generates a function with the prototype

```
void gen_x(Inst **ctp, a_type a, c_type c)
```

The **ctp** argument points to a pointer to the next instruction. ***ctp** is increased by the generation functions; i.e., you should allocate memory for the code to be generated beforehand, and start with ***ctp** set at the start of this memory area. Before running out of memory, allocate a new area, and generate a VM-level jump to the new area (this overflow handling is not implemented in our examples).

The other arguments correspond to the immediate arguments of the VM instruction (with their appropriate types as defined in the **type_prefix** declaration.

The following types, variables, and functions are used in **name-gen.i**:

Inst The type of the VM instruction; if you use threaded code, this is **void *;** for switch dispatch this is an integer type.

vm_prim The VM instruction table (type: **Inst ***, see VM instruction table).

gen_inst(Inst **ctp, Inst i) This function compiles the instruction i. Take a look at it in **vmgen-ex/peephole.c**. It is trivial when you don't want to use superinstructions (just the last two lines of the example function), and slightly more complicated in the example due to its ability to use superinstructions (see Peephole optimization).

genarg_type_prefix(Inst **ctp, type type_prefix) This compiles an immediate argument of type (as defined in a **type-prefix** definition). These functions are trivial to define (see **vmgen-ex/support.c**). You need one of these functions for every type that you use as immediate argument.

In addition to using these functions to generate code, you should call **BB_BOUNDARY** at every basic block entry point if you ever want to use superinstructions (or if you want to use the profiling supported by Vmgen; but this support is also useful mainly for selecting superinstructions). If you use **BB_BOUNDARY**, you should also define it (take a look at its definition in **vmgen-ex/mini.y**).

You do not need to call **BB_BOUNDARY** after branches, because you will not define superinstructions that contain branches in the middle (and if you did, and it would work, there would be no reason to end the superinstruction at the branch), and because the branches announce themselves to the profiler.

1.8.4 Peephole optimization

You need peephole optimization only if you want to use superinstructions. But having the code for it does not hurt much if you do not use superinstructions.

A simple greedy peephole optimization algorithm is used for superinstruction selection: every time **gen_inst** compiles a VM instruction, it checks if it can combine it with the last VM instruction (which may also be a superinstruction resulting from a previous peephole optimization); if so, it changes the last instruction to the combined instruction instead of laying down i at the current '***ctp**'.

The code for peephole optimization is in **vmgen-ex/peephole.c**. You can use this file almost verbatim. Vmgen generates **file-peephole.i** which contains data for the peephole optimizer.

You have to call '**init_peeptable()**' after initializing '**vm_prim**', and before compiling any VM code to initialize data structures for peephole optimization. After that, compiling with the VM code generation functions will automatically combine VM instructions into superinstructions. Since you do not want to combine instructions across VM branch targets (otherwise there will not be a proper VM instruction to branch to), you have to call **BB_BOUNDARY** (see VM code generation) at branch targets.

1.8.5 VM disassembler

A VM code disassembler is optional for an interpretive system, but highly recommended during its development and maintenance, because it is very useful for detecting bugs in the front end (and for distinguishing them from VM interpreter bugs).

Vmgen supports VM code disassembling by generating **file-disasm.i**. This code has to be wrapped into a function, as is done in **vmgen-ex/disasm.c**. You can use this file almost verbatim. In addition to '**vm_A2B(a,b)**', '**vm_out**', '**printarg_type(value)**', which are explained above, the following macros and variables are used in file-disasm.i (and you have to define them):

ip This variable points to the opcode of the current VM instruction.

IP IPTOS '**IPTOS**' is the first argument of the current VM instruction, and '**IP**' points to it; this is just as in the engine, but here '**ip**' points to the opcode of the VM instruction (in contrast to the engine, where '**ip**' points to the next cell, or even one further).

VM_IS_INST(Inst i, int n) Tests if the opcode '**i**' is the same as the 'n'th entry in the VM instruction table.

1.8.6 VM profiler

The VM profiler is designed for getting execution and occurence counts for VM instruction sequences, and these counts can then be used for selecting sequences as superinstructions. The VM profiler is probably not useful as profiling tool for the interpretive system. I.e., the VM profiler is useful for the developers, but not the users of the interpretive system.

The output of the profiler is: for each basic block (executed at least once), it produces the dynamic execution count of that basic block and all its subsequences; e.g.,

```
9227465 lit storelocal
9227465 storelocal branch
9227465 lit storelocal branch
```

I.e., a basic block consisting of '**lit storelocal branch**' is executed 9227465 times.

This output can be combined in various ways. E.g., **vmgen-ex/stat.awk** adds up the occurences of a given sequence wrt dynamic execution, static occurence, and per-program occurence. E.g.,

```
2 16 36910041 loadlocal lit
```

indicates that the sequence '**loadlocal lit**' occurs in 2 programs, in 16 places, and has been executed 36910041 times. Now you can select superinstructions in any way you like (note that compile time and space typically limit the number of superinstructions to 100–1000). After you have done that, **vmgen/seq2rule.awk** turns lines of the form above into rules for inclusion in a Vmgen input file. Note that this script does not ensure that all prefixes are defined, so you have to do that in other ways. So, an overall script for turning profiles into superinstructions can look like this:

```
awk -f stat.awk fib.prof test.prof|
awk '$3>=10000'| #select sequences
fgrep -v -f peephole-blacklist| #eliminate wrong instructions
awk -f seq2rule.awk| #turn into superinstructions
sort -k 3 >mini-super.vmg #sort sequences
```

Here the dynamic count is used for selecting sequences (preliminary results indicate that the static count gives better results, though); the third line eliminates sequences containing instructions that must not occur in a superinstruction, because they access a stack directly. The dynamic count selection ensures that all subsequences (including prefixes) of longer sequences occur (because subsequences have at least the same count as the longer sequences); the sort in the last line ensures that longer superinstructions occur after their prefixes.

But before using this, you have to have the profiler. Vmgen supports its creation by generating **`file-profile.i`**; you also need the wrapper file **`vmgen-ex/profile.c`** that you can use almost verbatim.

The profiler works by recording the targets of all VM control flow changes (through **`SUPER_END`** during execution, and through **`BB_BOUNDARY`** in the front end), and counting (through **`SUPER_END`**) how often they were targeted. After the program run, the numbers are corrected such that each VM basic block has the correct count (entering a block without executing a branch does not increase the count, and the correction fixes that), then the subsequences of all basic blocks are printed. To get all this, you just have to define **`SUPER_END`** (and **`BB_BOUNDARY`**) appropriately, and call **`vm_print_profile(FILE *file)`** when you want to output the profile on file.

The **`file-profile.i`** is similar to the disassembler file, and it uses variables and functions defined in **`vmgen-ex/profile.c`**, plus VM_IS_INST already defined for the VM disassembler (see VM disassembler).

1.9 Hints

- Floating point: and stacks

1.9.1 Floating point

How should you deal with floating point values? Should you use the same stack as for integers/pointers, or a different one? This section discusses this issue with a view on execution speed.

The simpler approach is to use a separate floating-point stack. This allows you to choose FP value size without considering the size of the integers/pointers, and you avoid a number of performance problems. The main downside is that this needs an FP stack pointer (and that may not fit in the register file on the 386 arhitecture, costing some performance, but comparatively little if you take the other option into account). If you use a separate FP stack (with stack pointer **`fp`**), using an **`fpTOS`** is helpful on most machines, but some spill the **`fpTOS`** register into memory, and **`fpTOS`** should not be used there.

The other approach is to share one stack (pointed to by, say, sp) between integer/pointer and floating-point values. This is ok if you do not use **`spTOS`**. If you do use **`spTOS`**, the compiler has to decide whether to put that variable into an integer or a floating point register, and the other type of operation becomes quite expensive on most machines (because moving values between integer and FP registers is quite expensive). If a value of one type has to be synthesized out of two values of the other type (double types), things are even more interesting.

One way around this problem would be to not use the **`spTOS`** supported by Vmgen, but to use explicit top-of-stack variables (one for integers, one for FP values), and having a kind of accumulator+stack architecture (e.g., Ocaml bytecode uses this approach); however, this is a major change, and it's ramifications are not completely clear.

1.10 The future

We have a number of ideas for future versions of Vmgen. However, there are so many possible things to do that we would like some feedback from you. What are you doing with Vmgen, what features are you missing, and why?

One idea we are thinking about is to generate just one .c file instead of letting you copy and adapt all the wrapper files (you would still have to define stuff like the type-specific macros, and stack pointers etc. somewhere). The advantage would be that, if we change the wrapper files between versions, you would not need to integrate your changes and our changes to them; Vmgen would also be easier to use for beginners. The main disadvantage of that is that it would reduce the flexibility of Vmgen a little (well, those who like flexibility could still patch the resulting .c file, like they are now doing for the wrapper files). In any case, if you are doing things to the wrapper files that would cause problems in a generated-.c-file approach, please let us know.

1.11 Contact

To report a bug, use `https://savannah.gnu.org/bugs/?func=addbug&group_id=2672`.

For discussion on Vmgen (e.g., how to use it), use the mailing list bug-vmgen@mail.freesoftware.fsf.org (use `http://mail.gnu.org/mailman/listinfo/help-vmgen` to subscribe).

You can find vmgen information at `http://www.complang.tuwien.ac.at/anton/vmgen/`.

2 Virtual machine implementation

2.1 ArrayForth application files

2.1.1 README

33 ⟨*README* 33⟩≡

```
This directory contains a working example for using vmgen.  It's a
small Modula-2-like programming language.

You can build the example by first installing Gforth and then saying,
in this directory:

make

Ignore the warnings.  You can check that it works with

make check

You can run mini programs like this:

./mini fib.mini

To learn about the options, type

./mini -h
```

```
More information can be found in the vmgen documentation.

-----
```
⟨*copyright* 64b⟩

2.1.2 simple.mini - example mini program

34a ⟨*simple.mini* 34a⟩≡
```
func main()
  return 1;
end func;
```

2.1.3 fib.mini - example mini program

34b ⟨*fib.mini* 34b⟩≡
```
func fib(n)
  var r;
  if n<2 then
    r:=1;
  else
    r:=fib(n-1)+fib(n-2);
  end if;
  return r;
  // the language syntax (return only at end) leads to inefficient code here
end func;

func main()
  return fib(34);
end func;
```

2.1.4 test.mini - example mini program (tests everything)

35 ⟨*test.mini* 35⟩≡

```
func operators()
  print 3 = 3;
  print (3+5) = 8;
  print (5-3) = 2;
  print (3*5) = 15;
  print (3&5) = 1;
  print (3|5) = 7;
  print (3<5) = 1;
  print (5<3) = 0;
  print (3=5) = 0;
  print (5=5) = 1;
  print (!3) = 0;
  print (!0) = 1;
  print (-3) = (0-3);
  return 0;
end func;

func params(a, b, c)
  print a = 3;
  print b = 5;
  print c = 7;
  return 9;
end func;

func locals(a)
  var b;
  var c;
  b:=a+1;
  c:=b+1;
  a:=c+1;
  return a;
end func;

func inc(x)
  return x+1;
end func;

func sign(n)
  var r;
  if (n<0) then
    r:=-1;
  else
    if (0<n) then
```

```
      r:=1;
    else
      r:=0;
    end if;
  end if;
  return r;
end func;

func recfac(n)
  var r;
  if (n<1) then
    r:=1;
  else
    r:=recfac(n-1)*n;
  end if;
  return r;
end func;

func itfac(n)
  var r;
  r:=1;
  while (0<n) do
    r:=r*n;
    n:=n-1;
  end while;
  return r;
end func;

func testfac()
  var i;
  i:=0;
  while (i<10) do
    print itfac(i) = recfac(i);
    i:=i+1;
  end while;
  return 0;
end func;

func main()
  operators();
  print params(3,5,7) = 9;
  print locals(3) = 6;
  print (inc(1)+inc(inc(inc(3))))=8;
  print sign(5) = 1;
  print sign(0) = 0;
  print sign(-5) = (-1);
```

```
  print itfac(5) = 120;
  testfac();
  return 0;
end func;
```

test.out - test.mini output

37 ⟨*test.out* 37⟩≡

```
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
result = 0
```

2.2 Mini specific files

You would typically change much in or replace the following files:

2.2.1 *Makefile*

38 ⟨*Makefile* 38⟩≡

```
#Makefile for vmgen example

⟨make-copyright 65b⟩

LEX=flex -l
YACC=bison -y
#YACC=yacc
VMGEN=vmgen
#GCC=gcc -g -Wall
GCC=gcc -O3 -fomit-frame-pointer -Wall
CC=$(GCC)
#M4=m4 -s #recommended if supported
M4=m4
OBJECTS_MINI=mini.tab.o support.o peephole.o profile.o disasm.o engine.o engine-debug.o

mini: $(OBJECTS_MINI)
        $(CC) $(OBJECTS_MINI) -o $@

lex.yy.c: mini.l
        $(LEX) mini.l

mini.tab.c: mini.y lex.yy.c
        $(YACC) mini.y && mv y.tab.c $@

mini-vm.i mini-disasm.i mini-gen.i mini-labels.i mini-profile.i mini-peephole.i: mini.vmg
        $(VMGEN) mini.vmg

mini.vmg: mini-inst.vmg mini-super.vmg
        $(M4) mini-inst.vmg >$@

mini.tab.o: mini.tab.c mini-gen.i lex.yy.c mini.h

support.o: support.c mini.h

peephole.o: peephole.c mini-peephole.i mini.h

profile.o: profile.c mini-profile.i mini.h

disasm.o: disasm.c mini-disasm.i mini.h
```

⟨*engine* 60⟩

```
clean:
        rm -f *.o mini mini-*.i lex.yy.c mini.tab.c mini.vmg

check:  mini
        ./mini test.mini | tr -d '\015' | diff - test.out

checkall:
        for i in 1 3 5 8 9 10; do make clean; echo $$i; make check VMGEN=vmgen CC="gcc -O3

#make profiles
%.prof: %.mini mini
        ./mini -p $< 2>$@
```

2.2.2 Support.c - main() and other support functions

40 ⟨*support.c* 40⟩≡

```
/* support functions and main() for vmgen example

⟨copyright 64b⟩
*/

#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
extern int optind;

#include <assert.h>
#include "mini.h"

void genarg_i(Inst **vmcodepp, Cell i)
{
  *((Cell *) *vmcodepp) = i;
  (*vmcodepp)++;
}

void genarg_target(Inst **vmcodepp, Inst *target)
{
  *((Inst **) *vmcodepp) = target;
  (*vmcodepp)++;
}

void printarg_i(Cell i)
{
  fprintf(vm_out, "%ld ", i);
}

void printarg_target(Inst *target)
{
  fprintf(vm_out, "%p ", target);
}

void printarg_a(char *a)
{
  fprintf(vm_out, "%p ", a);
}

void printarg_Cell(Cell i)
{
  fprintf(vm_out, "0x%lx ", i);
```

```
}

/* This language has separate name spaces for functions and variables;
   this works because there are no function variables, and the syntax
   makes it possible to differentiate between function and variable
   reference */

typedef struct functab {
  struct functab *next;
  char *name;
  Inst *start;
  int params;
  int nonparams;
} functab;

functab *ftab=NULL;

/* note: does not check for double definitions */
void insert_func(char *name, Inst *start, int locals, int nonparams)
{
  functab *node = malloc(sizeof(functab));

  node->next=ftab;
  node->name=name;
  node->start=start;
  node->params=locals-nonparams;
  node->nonparams=nonparams;
  ftab=node;
}

functab *lookup_func(char *name)
{
  functab *p;

  for (p=ftab; p!=NULL; p=p->next)
    if (strcmp(p->name,name)==0)
      return p;
  fprintf(stderr, "undefined function %s", name);
  exit(1);
}

Inst *func_addr(char *name)
{
  return lookup_func(name)->start;
}
```

```
Cell func_calladjust(char *name)
{
  return adjust(lookup_func(name)->nonparams);
}

typedef struct vartab {
  struct vartab *next;
  char *name;
  int index;
} vartab;

vartab* vtab;

/* no checking for double definitions */
void insert_local(char *name)
{
  vartab *node = malloc(sizeof(vartab));

  locals++;
  node->next=vtab;
  node->name=name;
  node->index=locals;
  vtab = node;
}

vartab *lookup_var(char *name)
{
  vartab *p;

  for (p=vtab; p!=NULL; p=p->next)
    if (strcmp(p->name,name)==0)
      return p;
  fprintf(stderr, "undefined local variable %s", name);
  exit(1);
}

Cell var_offset(char *name)
{
  return (locals - lookup_var(name)->index + 2)*sizeof(Cell);
}

#define CODE_SIZE 65536
#define STACK_SIZE 65536
typedef Cell (*engine_t)(Inst *ip0, Cell* sp, char* fp);

char *program_name;
```

```
int main(int argc, char **argv)
{
  int disassembling = 0;
  int profiling = 0;
  int c;
  Inst *vm_code=(Inst *)calloc(CODE_SIZE,sizeof(Inst));
  Inst *start;
  Cell *stack=(Cell *)calloc(STACK_SIZE,sizeof(Cell));
  engine_t runvm=engine;

  while ((c = getopt(argc, argv, "hdpt")) != -1) {
    switch (c) {
    default:
    case 'h':
    help:
      fprintf(stderr, "Usage: %s [options] file\nOptions:\n-h   Print this message and exit
              argv[0]);
      exit(1);
    case 'd':
      disassembling=1;
      break;
    case 'p':
      profiling=1;
      use_super=0; /* we don't want superinstructions in the profile */
      runvm = engine_debug;
      break;
    case 't':
      vm_debug=1;
      runvm = engine_debug;
      break;
    }
  }
  if (optind+1 != argc)
    goto help;
  program_name = argv[optind];
  if ((yyin=fopen(program_name,"r"))==NULL) {
    perror(argv[optind]);
    exit(1);
  }

  /* initialize everything */
  vmcodep = vm_code;
  vm_out = stderr;
  (void)runvm(NULL,NULL,NULL); /* initialize vm_prim */
  init_peeptable();
```

```
  if (yyparse())
    exit(1);

  start=vmcodep;
  gen_main_end();
  vmcode_end=vmcodep;

  if (disassembling)
    vm_disassemble(vm_code, vmcodep, vm_prim);

  printf("result = %ld\n",runvm(start, stack+STACK_SIZE-1, NULL));

  if (profiling)
    vm_print_profile(vm_out);

  return 0;
}
```

2.2.3 mini.h - common declarations

45 ⟨*mini.h* 45⟩≡

```
/* support functions for vmgen example

⟨copyright 64b⟩
*/

#include <stdio.h>

typedef long Cell;
#ifdef __GNUC__
typedef void *Label;
typedef Label Inst; /* we could "typedef Cell Inst", removing the need
                       for casts in a few places, but requiring a few
                       casts etc. in other places */
#else
typedef long Label;
typedef long Inst;
#endif

extern Inst *vm_prim;
extern int locals;
extern Cell peeptable;
extern int vm_debug;
extern FILE *yyin;
extern int yylineno;
extern char *program_name;
extern FILE *vm_out;
extern Inst *vmcodep;
extern Inst *last_compiled;
extern Inst *vmcode_end;
extern int use_super;

/* generic vmgen support functions (e.g., wrappers) */
void gen_inst(Inst **vmcodepp, Inst i);
void init_peeptable(void);
void vm_disassemble(Inst *ip, Inst *endp, Inst prim[]);
void vm_count_block(Inst *ip);
struct block_count *block_insert(Inst *ip);
void vm_print_profile(FILE *file);

⟨vm-A2B(a,b) 26a⟩

⟨IMM-ARG 27a⟩
```

```
#define VM_IS_INST(inst, n) ((inst) == vm_prim[n])

/* mini type-specific support functions */
void genarg_i(Inst **vmcodepp, Cell i);
void printarg_i(Cell i);
void genarg_target(Inst **vmcodepp, Inst *target);
void printarg_target(Inst *target);
void printarg_a(char *a);
void printarg_Cell(Cell i);

/* engine functions (type not fixed) */
Cell engine(Inst *ip0, Cell *sp, char *fp);
Cell engine_debug(Inst *ip0, Cell *sp, char *fp);

/* other generic functions */
int yyparse(void);

/* mini-specific functions */
void insert_func(char *name, Inst *start, int locals, int nonparams);
Inst *func_addr(char *name);
Cell func_calladjust(char *name);
void insert_local(char *name);
Cell var_offset(char *name);
void gen_main_end(void);

/* stack pointer change for a function with n nonparams */
#define adjust(n)  ((n) * -sizeof(Cell))
```

2.2.4 mini-inst.vmg - simple VM instructions

47a ⟨*mini-inst.vmg* 47a⟩≡

```
\ mini.inst is generated automatically from mini-inst.vmg and mini-super.vmg
\ example .vmg file
```

⟨© 65a⟩

```
\ WARNING: This file is processed by m4. Make sure your identifiers
\ don't collide with m4's (e.g. by undefining them).

\ comments start with "\ "
```

⟨*stack-definitions* 11b⟩

⟨*stack-prefix* 11d⟩

⟨*type-prefix* 11a⟩

⟨*simple-instructions* 10⟩

⟨*branch* 12b⟩

⟨*zbranch* 13b⟩

⟨*stack-organization* 15b⟩

⟨*stack-caching* 14⟩

⟨*end* 13a⟩

```
include(mini-super.vmg)
```

2.2.5 mini-super.vmg - superinstructions (empty at first)

47b ⟨*mini-super.vmg* 47b⟩≡

⟨*ll* 16⟩

2.2.6 mini.l - scanner

48 ⟨*mini.l* 48⟩≡

```
%{
/* front-end scanner for vmgen example

⟨copyright 64b⟩
*/

/* % option yylineno (flex option, implied by flex -l) */

#include <stdlib.h>
#include <string.h>
char *mystrdup(const char *s)
{
  char *t=malloc(strlen(s)+1);
  return strcpy(t,s);
}
%}

%%
[-();,+*&|<=!]  return yytext[0];
:=     return BECOMES;
func   return FUNC;
return return RETURN;
end    return END;
var    return VAR;
if     return IF;
then   return THEN;
else   return ELSE;
while  return WHILE;
do     return DO;
print  return PRINT;
[0-9]+ { yylval.num=strtol(yytext,NULL,10); return NUM; }
[a-zA-Z\_][a-zA-Z0-9\_]* { yylval.string=mystrdup(yytext); return IDENT; }
[ \t\n] ;
[/][/].* ;
. yyerror("illegal character"); exit(1);
%%
```

2.2.7 mini.y - front end (parser, VM code generator)

49 ⟨*mini.y* 49⟩≡

```
/* front-end compiler for vmgen example

⟨copyright 64b⟩
*/

/* I use yacc/bison here not because I think it's the best tool for
   the job, but because it's widely available and popular; it's also
   (barely) adequate for this job. */

%{
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mini.h"

/* BB_BOUNDARY is needed on basic blocks without a preceding VM branch */
#define BB_BOUNDARY (last_compiled = NULL,  /* suppress peephole opt */ \
                     block_insert(vmcodep)) /* for accurate profiling */

Inst *vm_prim;
Inst *vmcodep;
FILE *vm_out;
int vm_debug;

void yyerror(char *s)
{
#if 1
  /* for pure flex call */
  fprintf(stderr, "%s: %s\n", program_name, s);
#else
  /* lex or flex -l supports yylineno */
  fprintf (stderr, "%s: %d: %s\n", program_name, yylineno, s);
#endif
}

#include "mini-gen.i"

void gen_main_end(void)
{
  gen_call(&vmcodep, func_addr("main"), func_calladjust("main"));
  gen_end(&vmcodep);
  BB_BOUNDARY; /* for profiling; see comment in mini.vmg:end */
}
```

```
int locals=0;
int nonparams=0;

int yylex();
%}

%token FUNC RETURN END VAR IF THEN ELSE WHILE DO BECOMES PRINT NUM IDENT

%union {
  long num;
  char *string;
  Inst *instp;
}

%type <string> IDENT;
%type <num> NUM;
%type <instp> elsepart;

%%
program: program function
       | ;

function: FUNC IDENT { locals=0; nonparams=0; } '(' params ')'
          vars                  { insert_func($2,vmcodep,locals,nonparams); }
          stats RETURN expr ';'
          END FUNC ';'          { gen_return(&vmcodep, -adjust(locals)); }
        ;

params: IDENT ',' { insert_local($1); } params
      | IDENT     { insert_local($1); }
      | ;

vars: vars VAR IDENT ';' { insert_local($3); nonparams++; }
    | ;

stats: stats stat ';'
     | ;

stat: IF expr THEN { gen_zbranch(&vmcodep, 0); $<instp>$ = vmcodep; }
      stats { $<instp>$ = $<instp>4; }
      elsepart END IF { BB_BOUNDARY; $<instp>7[-1] = (Inst)vmcodep; }
    | WHILE   { BB_BOUNDARY; $<instp>$ = vmcodep; }
      expr DO { gen_zbranch(&vmcodep, 0); $<instp>$ = vmcodep; }
      stats END WHILE { gen_branch(&vmcodep, $<instp>2); $<instp>5[-1] = (Inst)vmcodep; }
    | IDENT BECOMES expr        { gen_storelocal(&vmcodep,  var_offset($1)); }
```

```
    | PRINT expr                { gen_print(&vmcodep); }
    | expr                      { gen_drop(&vmcodep); }
    ;

elsepart: ELSE { gen_branch(&vmcodep, 0); $<instp>$ = vmcodep; $<instp>0[-1] = (Inst)vmcode
          stats { $$ = $<instp>2; }
        | { $$ = $<instp>0; }
        ;

expr: term '+' term     { gen_add(&vmcodep); }
    | term '-' term     { gen_sub(&vmcodep); }
    | term '*' term     { gen_mul(&vmcodep); }
    | term '&' term     { gen_and(&vmcodep); }
    | term '|' term     { gen_or(&vmcodep); }
    | term '<' term     { gen_lessthan(&vmcodep); }
    | term '=' term     { gen_equals(&vmcodep); }
    | '!' term          { gen_not(&vmcodep); }
    | '-' term          { gen_negate(&vmcodep); }
    | term
    ;

term: '(' expr ')'
    | IDENT '(' args ')' { gen_call(&vmcodep, func_addr($1), func_calladjust($1)); }
    | IDENT              { gen_loadlocal(&vmcodep, var_offset($1)); }
    | NUM                { gen_lit(&vmcodep, $1); }
    ;

/* missing: argument counting and checking against called function */
args: expr ',' args
    | expr
    | ;
%%
int yywrap(void)
{
  return 1;
}

#include "lex.yy.c"
```

2.2.8 peephole-blacklist - list of instructions not allowed in superinstructions

51 ⟨*peephole-blacklist* 51⟩≡

```
call
return
unknown
```

2.3 Generic support files

For your own interpreter, you would typically copy the following files and change little, if anything:

2.3.1 peephole.c - wrapper file

53 ⟨*peephole.c* 53⟩≡

```
/* Peephole optimization routines and tables

⟨copyright 64b⟩
*/

#include <stdlib.h>
#include "mini.h"

/* the numbers in this struct are primitive indices */
typedef struct Combination {
  int prefix;
  int lastprim;
  int combination_prim;
} Combination;

Combination peephole_table[] = {
#include "mini-peephole.i"
#ifndef __GNUC__
  {-1,-1,-1} /* unnecessary; just to shut up lcc if the file is empty */
#endif
};

int use_super = 1; /* turned off by option -p */

typedef struct Peeptable_entry {
  struct Peeptable_entry *next;
  Inst prefix;
  Inst lastprim;
  Inst combination_prim;
} Peeptable_entry;

#define HASH_SIZE 1024
#define hash(_i1,_i2) (((((Cell)(_i1))^((Cell)(_i2)))»4)&(HASH_SIZE-1))

Cell peeptable;

Cell prepare_peephole_table(Inst insts[])
{
  Cell i;
```

```
  Peeptable_entry **pt = (Peeptable_entry **)calloc(HASH_SIZE,sizeof(Peeptable_entry *));

  for (i=0; i<sizeof(peephole_table)/sizeof(peephole_table[0]); i++) {
    Combination *c = &peephole_table[i];
    Peeptable_entry *p = (Peeptable_entry *)malloc(sizeof(Peeptable_entry));
    Cell h;
    p->prefix =           insts[c->prefix];
    p->lastprim =         insts[c->lastprim];
    p->combination_prim = insts[c->combination_prim];
    h = hash(p->prefix,p->lastprim);
    p->next = pt[h];
    pt[h] = p;
  }
  return (Cell)pt;
}

void init_peeptable(void)
{
  peeptable = prepare_peephole_table(vm_prim);
}

Inst peephole_opt(Inst inst1, Inst inst2, Cell peeptable)
{
  Peeptable_entry **pt = (Peeptable_entry **)peeptable;
  Peeptable_entry *p;

  if (use_super == 0)
      return 0;
  for (p = pt[hash(inst1,inst2)]; p != NULL; p = p->next)
    if (inst1 == p->prefix && inst2 == p->lastprim)
      return p->combination_prim;
  return NULL;
}

Inst *last_compiled = NULL;

void gen_inst(Inst **vmcodepp, Inst i)
{
  if (last_compiled != NULL) {
    Inst combo = peephole_opt(*last_compiled, i, peeptable);
    if (combo != NULL) {
      *last_compiled = combo;
      return;
    }
  }
  last_compiled = *vmcodepp;
```

```
    **vmcodepp = i;
    (*vmcodepp)++;
}
```

2.3.2 profile.c - wrapper file

56 ⟨*profile.c* 56⟩≡

```
/* VM profiling support stuff

⟨copyright 64b⟩
*/

#include <stdlib.h>
#include <stdio.h>
#include <assert.h>
#include "mini.h"

/* data structure: simple hash table with external chaining */

#define HASH_SIZE (1«20)
#define hash(p) ((((Cell)(p))/sizeof(Inst))&(HASH_SIZE-1))

#ifdef __GNUC__
typedef long long long_long;
#else
typedef long long_long;
#endif

typedef struct block_count {
  struct block_count *next; /* next in hash table */
  struct block_count *fallthrough; /* the block that this one falls
                                      through to without SUPER_END */
  Inst *ip;
  long_long count;
  char **insts;
  size_t ninsts;
} block_count;

block_count *blocks[HASH_SIZE];
Inst *vmcode_end;

block_count *block_lookup(Inst *ip)
{
  block_count *b = blocks[hash(ip)];

  while (b!=NULL && b->ip!=ip)
    b = b->next;
  return b;
}
```

```
/* looks up present elements, inserts absent elements */
block_count *block_insert(Inst *ip)
{
  block_count *b = block_lookup(ip);
  block_count *new;

  if (b != NULL)
    return b;
  new = (block_count *)malloc(sizeof(block_count));
  new->next = blocks[hash(ip)];
  new->fallthrough = NULL;
  new->ip = ip;
  new->count = (long_long)0;
  new->insts = malloc(1);
  assert(new->insts != NULL);
  new->ninsts = 0;
  blocks[hash(ip)] = new;
  return new;
}

void add_inst(block_count *b, char *inst)
{
  b->insts = realloc(b->insts, (b->ninsts+1) * sizeof(char *));
  b->insts[b->ninsts++] = inst;
}

void vm_count_block(Inst *ip)
{
  block_insert(ip)->count++;
}

void postprocess_block(block_count *b)
{
  Inst *ip = b->ip;
  block_count *next_block=NULL;

  while (next_block == NULL && ip<vmcode_end) {
#include "mini-profile.i"
    /* else */
    {
      add_inst(b,"unknown");
      ip++;
    }
  _endif_:
    next_block = block_lookup(ip);
  }
```

```
  /* we fell through, so set fallthrough and update the count */
  b->fallthrough = next_block;
  /* also update the counts of all following fallthrough blocks that
     have already been processed */
  while (next_block != NULL) {
    next_block->count += b->count;
    next_block = next_block->fallthrough;
  }
}

/* Deal with block entry by falling through from non-SUPER_END
   instructions.  And fill the insts and ninsts fields. */
void postprocess(void)
{
  size_t i;

  for (i=0; i<HASH_SIZE; i++) {
    block_count *b = blocks[i];
    for (; b!=0; b = b->next)
      postprocess_block(b);
   }
}

#if 0
void print_block(FILE *file, block_count *b)
{
  size_t i;

  fprintf(file,"%14lld\t",b->count);
  for (i=0; i<b->ninsts; i++)
    fprintf(file, "%s ", b->insts[i]);
  putc('\n', file);
}
#endif

void print_block(FILE *file, block_count *b)
{
  size_t i,j,k;

  for (i=2; i<12; i++)
    for (j=0; i+j<=b->ninsts; j++) {
      fprintf(file,"%14lld\t",b->count);
      for (k=j; k<i+j; k++)
        fprintf(file, "%s ", b->insts[k]);
      putc('\n', file);
    }
```

```
}

void vm_print_profile(FILE *file)
{
  size_t i;

  postprocess();
  for (i=0; i<HASH_SIZE; i++) {
    block_count *b = blocks[i];
    for (; b!=0; b = b->next)
      print_block(file, b);
   }
}
```

2.3.3 disasm.c - wrapper file

59 ⟨*disasm.c* 59⟩≡

```
/* vm disassembler wrapper

⟨copyright 64b⟩
*/

#include "mini.h"

#define IP (ip+1)
#define IPTOS IP[0]

void vm_disassemble(Inst *ip, Inst *endp, Inst vm_prim[])
{
  while (ip<endp) {
    fprintf(vm_out,"%p: ",ip);
#include "mini-disasm.i"
    {
      fprintf(vm_out,"unknown instruction %p",ip[0]);
      ip++;
    }
  _endif_:
    fputc('\n',vm_out);
  }
}
```

2.3.4 engine.c - wrapper file

60 ⟨*engine* 60⟩≡ (38)

```
engine.o: engine.c mini-vm.i mini-labels.i mini.h

engine-debug.o: engine.c mini-vm.i mini-labels.i mini.h
        $(CC) -DVM_DEBUG -DVM_PROFILING -Dengine=engine_debug -c -o $@ engine.c
```

61 ⟨*engine.c* 61⟩≡

```
/* vm interpreter wrapper

⟨copyright 64b⟩
*/

#include "mini.h"

⟨IF-spTOS 27d⟩

⟨NAME 23c⟩

⟨dispatch 20⟩

⟨LABEL2 23b⟩

⟨SUPER-END 27e⟩

#ifndef __GNUC__
enum {
#include "mini-labels.i"
};
#endif

⟨MAYBE-UNUSED 28a⟩

/* the return type can be anything you want it to */
Cell engine(Inst *ip0, Cell *sp, char *fp)
{
  /* VM registers (you may want to use gcc's "Explicit Reg Vars" here) */
  Inst * ip;
  Inst * cfa;
⟨spTOS 27b⟩
  static Label labels[] = {
#include "mini-labels.i"
  };
#ifdef MORE_VARS
  MORE_VARS
#endif

  if (vm_debug)
      fprintf(vm_out,"entering engine(%p,%p,%p)\n",ip0,sp,fp);
  if (ip0 == NULL) {
    vm_prim = labels;
    return 0;
  }
```

```
    /* I don't have a clue where these things come from,
       but I've put them in macros.h for the moment */
    IF_spTOS(spTOS = sp[0]);

    SET_IP(ip0);
    SUPER_END;  /* count the BB starting at ip0 */

  #ifdef __GNUC__
    NEXT;
  #include "mini-vm.i"
  #else
   next_inst:
    switch(*ip++) {
  #include "mini-vm.i"
    default:
      fprintf(stderr,"unknown instruction %d at %p\n", ip[-1], ip-1);
      exit(1);
    }
  #endif
  }
```

2.3.5 stat.awk - script for aggregating profile information

62 ⟨*stat.awk* 62⟩≡

```
BEGIN {
  FS="\t";
}
{
  dyn[$2] += $1;
  stat[$2]++;
  files[$2] += (FILENAME!=filename[$2]);
  filename[$2] = FILENAME;
}
END {
  for (i in dyn)
    printf("%7d\t%7d\t%15d\t%s\n",files[i],stat[i],dyn[i],i);
}
```

2.3.6 seq2rule.awk - script for creating superinstructions

63a ⟨*seq2rule.awk* 63a⟩≡

```
BEGIN {
  FS="\t";
}
{
  name = $4;
  gsub(/ /,"_",name);
  print name" = "$4;
}
```

3 Printing and Extracting the code

A script for converting this document to PDF form follows:

63b ⟨*final* 63b⟩≡

```
lyx -e pdf $1
lyx -e latex $1
```

Each of these scripts can be pulled out manually given the default * script defined below.

63c ⟨* 63c⟩≡

```
echo "Extract file $1 from afvm.lyx..."
rm -f afvm.nw
lyx -e literate afvm.lyx
notangle -R$2 afvm.nw > $1
chmod a+x $1
rm -f afvm.nw
```

Once that script is pulled out and named extract, the following script can pull out all of the other scripts:

64a ⟨*extract-all* 64a⟩≡

```
echo "Extract all files..."
./extract README
./extract simple.mini
./extract fib.mini
./extract test.mini
./extract test.out
./extract Makefile
./extract support.c
./extract mini.h
./extract mini-inst.vmg
./extract mini-super.vmg
./extract mini.l
./extract mini.y
./extract peephole-blacklist
./extract peephole.c
./extract profile.c
./extract disasm.c
./extract engine.c
./extract stat.awk
./extract seq2rule.awk
```

4 Copyrights

Although I would have preferred to use an MIT license, GForth uses a GNU license, so I must comply with that.

64b ⟨*copyright* 64b⟩≡ (33 40 45 48 49 53 56 59 61)

```
Copyright (C) 2001,2002,2003,2007 Free Software Foundation, Inc.

This file is part of Gforth.

Gforth is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation, either version 3
of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, see http://www.gnu.org/licenses/.
```

65a ⟨© 65a⟩≡ (47a)

```
\ Copyright (C) 2001,2002,2003,2007 Free Software Foundation, Inc.

\ This file is part of Gforth.

\ Gforth is free software; you can redistribute it and/or
\ modify it under the terms of the GNU General Public License
\ as published by the Free Software Foundation, either version 3
\ of the License, or (at your option) any later version.

\ This program is distributed in the hope that it will be useful,
\ but WITHOUT ANY WARRANTY; without even the implied warranty of
\ MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
\ GNU General Public License for more details.

\ You should have received a copy of the GNU General Public License
\ along with this program. If not, see http://www.gnu.org/licenses/.
```

65b ⟨*make-copyright* 65b⟩≡ (38)

```
#Copyright (C) 2001,2003,2007,2008 Free Software Foundation, Inc.

#This file is part of Gforth.

#Gforth is free software; you can redistribute it and/or
#modify it under the terms of the GNU General Public License
#as published by the Free Software Foundation, either version 3
#of the License, or (at your option) any later version.

#This program is distributed in the hope that it will be useful,
#but WITHOUT ANY WARRANTY; without even the implied warranty of
#MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.#See the
#GNU General Public License for more details.

#You should have received a copy of the GNU General Public License
#along with this program; if not, see http://www.gnu.org/licenses/.
```

4.1 GNU GENERAL PUBLIC LICENSE

Version 3, 29 June 2007

Copyright © 2007 Free Software Foundation, Inc. <http://fsf.org/>

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The GNU General Public License is a free, copyleft license for software and other kinds of works.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change all versions of a program--to make sure it remains free software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software; it applies also to any other work released this way by its authors. You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

Developers that use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS

Definitions.

"This License" refers to version 3 of the GNU General Public License.

"Copyright" also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

"The Program" refers to any copyrightable work licensed under this License. Each licensee is addressed as "you". "Licensees" and "recipients" may be individuals or organizations.

To “modify” a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a “modified version” of the earlier work or a work “based on” the earlier work.

A “covered work” means either the unmodified Program or a work based on the Program.

To “propagate” a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

To “convey” a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

An interactive user interface displays “Appropriate Legal Notices” to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion.

4.1.1 Source Code.

The “source code” for a work means the preferred form of the work for making modifications to it. “Object code” means any non-source form of a work.

A “Standard Interface” means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

The “System Libraries” of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A “Major Component”, in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

The “Corresponding Source” for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work’s System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work.

4.1.2 Basic Permissions.

All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The

output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 4.1.6 makes it unnecessary.

4.1.3 Protecting Users' Legal Rights From Anti-Circumvention Law.

No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures.

4.1.4 Conveying Verbatim Copies.

You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with section 4.1.5 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee.

4.1.5 Conveying Modified Source Versions.

You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4.1.4, provided that you also meet all of these conditions:

- a) The work must carry prominent notices stating that you modified it, and giving a relevant date.
- b) The work must carry prominent notices stating that it is released under this License and any conditions added under section 4.1.9. This requirement modifies the requirement in section 4.1.4 to "keep intact all notices".
- c) You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 4.1.9 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it.

- d) If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so. A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an “aggregate” if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation’s users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate.

4.1.6 Conveying Non-Source Forms.

You may convey a covered work in object code form under the terms of sections 4.1.4 and 4.1.5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

- a) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange.
- b) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge.
- c) Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 4.1.6b.
- d) Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements.
- e) Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 4.1.6d. A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A “User Product” is either (1) a “consumer product”, which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, “normally used” refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

"Installation Information" for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source. The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying.

4.1.7 Additional Terms.

"Additional permissions" are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

- a) Disclaiming warranty or limiting liability differently from the terms of sections 4.1.15 and 4.1.16 of this License; or
- b) Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or
- c) Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or
- d) Limiting the use for publicity purposes of names of licensors or authors of the material; or
- e) Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or
- f) Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered “further restrictions” within the meaning of section 4.1.10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way.

4.1.8 Termination.

You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 4.1.12).

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 4.1.10.

4.1.9 Acceptance Not Required for Having Copies.

You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance. However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so.

4.1.10 Automatic Licensing of Downstream Recipients.

Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

An “entity transaction” is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party’s predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it.

4.1.11 Patents.

A "contributor" is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor's "contributor version".

A contributor's "essential patent claims" are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, "control" includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor's essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

In the following three paragraphs, a "patent license" is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To "grant" such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. "Knowingly relying" means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient's use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is "discriminatory" if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law.

4.1.12 No Surrender of Others' Freedom.

If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program.

4.1.13 Use with the GNU Affero General Public License.

Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 4.1.12, concerning interaction through a network will apply to the combination as such.

4.1.14 Revised Versions of this License.

The Free Software Foundation may publish revised and/or new versions of the GNU General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version.

4.1.15 Disclaimer of Warranty.

THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

4.1.16 Limitation of Liability.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE,

BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

4.1.17 Interpretation of Sections 4.1.15 and 4.1.16.

If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively state the exclusion of warranty; and each file should have at least the “copyright” line and a pointer to where the full notice is found.

```
<one line to give the program's name and a brief idea of what it does.>
Copyright (C) <year> <name of author>

This program is free software:  you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program.  If not, see <http://www.gnu.org/licenses/>.
```

Also add information on how to contact you by electronic and paper mail.

If the program does terminal interaction, make it output a short notice like this when it starts in an interactive mode:

```
<program> Copyright (C) <year> <name of author>
This program comes with ABSOLUTELY NO WARRANTY; for details type ‘show w’.
This is free software, and you are welcome to redistribute it
under certain conditions; type ‘show c’ for details.
```

The hypothetical commands ‘show w’ and ‘show c’ should show the appropriate parts of the General Public License. Of course, your program’s commands might be different; for a GUI interface, you would use an “about box”.

You should also get your employer (if you work as a programmer) or school, if any, to sign a “copyright disclaimer” for the program, if necessary. For more information on this, and how to apply and follow the GNU GPL, see <http://www.gnu.org/licenses/>.

The GNU General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License. But first, please read <http://www.gnu.org/philosophy/why-not-lgpl.html>.

4.2 GNU Free Documentation License

Version 1.3, 3 November 2008

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

4.2.1 APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document’s overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary

then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

4.2.2 VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 4.2.3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

4.2.3 COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and

legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4.2.4 MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 4.2.2 and 4.2.3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
- C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
- D. Preserve all the copyright notices of the Document.
- E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document’s license notice.
- H. Include an unaltered copy of this License.
- I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

- J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
- N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
- O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

4.2.5 COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4.2.4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

4.2.6 COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

4.2.7 AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 4.2.3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

4.2.8 TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4.2.4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4.2.4) to Preserve its Title (section 4.2.1) will typically require changing the actual title.

4.2.9 TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

4.2.10 FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See `http://www.gnu.org/copyleft/`.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

4.2.11 RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C) YEAR YOUR NAME.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.3
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.
A copy of the license is included in the section entitled "GNU Free Documentation
License".
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with … Texts." line with this:

```
with the Invariant Sections being LIST THEIR TITLES, with the
Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.