Hobbit

SCM Compiler
Version 5f3

Tanel Tammet
Department of Computing Science
Chalmers University of Technology
University of Go"teborg
S-41296 Go"teborg Sweden

This manual is for the Hobbit compiler for SCM (version 5f3, February 2020),

Table of Contents

1 Introduction

Hobbit is a small optimizing scheme-to-C compiler written in Report 4 scheme and intended for use together with the SCM scheme interpreter of A. Jaffer. Hobbit compiles full Report 4 scheme, except that:

- It does not fully conform to the requirement of being properly tail-recursive: non-mutual tailrecursion is detected, but mutual tailrecursion is not.
- Macros from the Report 4 appendix are not supported (yet): only the common-lisp-like defmacro is supported.

Hobbit treats SCM files as a C library and provides integration of compiled procedures and variables with the SCM interpreter as new primitives.

Hobbit compiles scheme files to C files and does not provide anything else by itself (eg. calling the C compiler, dynamic loading). Such niceties are described in the next chapter Section 2.1 [Compiling And Linking], page 2.

Hobbit (derived from hobbit5x) is now part of the SCM Scheme implementation. The most recent information about SCM can be found on SCM's *WWW* home page:

`http://people.csail.mit.edu/jaffer/SCM`

Hobbit4d has also been ported to the Guile Scheme implementation:

`http://www.gnu.org/software/guile/anon-cvs.html`

2 Compiling with Hobbit

2.1 Compiling And Linking

```
(require 'compile)
```

`hobbit` *name1.scm name2.scm ...* [Function]

Invokes the HOBBIT compiler to translate Scheme files *name1*`.scm`, *name2*`.scm`, ... to C files *name1*`.c` and *name1*`.h`.

`compile-file` *name1.scm name2.scm ...* [Function]

Compiles the HOBBIT translation of *name1*.scm, *name2*.scm, ... to a dynamically linkable object file *name1*<object-suffix>, where <object-suffix> is the object file suffix for your computer (for instance, `.so`). *name1*.scm must be in the current directory; *name2*.scm, ... may be in other directories.

If a file named *name1*`.opt` exists, then its options are passed to the `build` invocation which compiles the `c` files.

```
cd ~/scm/
scm -rcompile -e"(compile-file \"example.scm\")"

Starting to read example.scm

Generic (slow) arithmetic assumed: 1.0e-3 found.

** Pass 1 completed **
** Pass 2 completed **
** Pass 3 completed **
** Pass 4 completed **
** Pass 5 completed **
** Pass 6 completed **

C source file example.c is built.
C header file example.h is built.

These top level higher order procedures are not clonable (slow):
(nonkeyword_make-promise map-streams generate-vector runge-kutta-4)
These top level procedures create non-liftable closures (slow):
(nonkeyword_make-promise damped-oscillator map-streams scale-vector elementwise runge-

; Scheme (linux) script created by SLIB/batch Sun Apr  7 22:49:49 2002
; ================ Write file with C defines
(delete-file "scmflags.h")
(call-with-output-file
  "scmflags.h"
  (lambda (fp)
    (for-each
```

```
        (lambda (string) (write-line string fp))
        '("#define IMPLINIT \"Init5f3.scm\""
          "#define BIGNUMS"
          "#define FLOATS"
          "#define ARRAYS"
          "#define DLL"))))
    ; ================ Compile C source files
    (system "gcc -O2 -fpic -c -I/usr/local/lib/scm/ example.c")
    (system "gcc -shared -o example.so example.o -lm -lc")
    (delete-file "example.o")
    ; ================ Link C object files
    (delete-file "slibcat")

    Compilation finished at Sun Apr  7 22:49:50
```

`compile->executable` *exename name1.scm name2.scm . . .* [Function]

Compiles and links the HOBBIT translation of *name1*.scm, *name2*.scm, . . . to a SCM executable named *exename*. *name1*.scm must be in the current directory; *name2*.scm, . . . may be in other directories.

If a file named *`exename`*`.opt` exists, then its options are passed to the `build` invocation which compiles the `c` files.

```
cd ~/scm/
scm -rcompile -e"(compile->executable \"exscm\" \"example.scm\")"

Starting to read example.scm

Generic (slow) arithmetic assumed: 1.0e-3 found.

** Pass 1 completed **
** Pass 2 completed **
** Pass 3 completed **
** Pass 4 completed **
** Pass 5 completed **
** Pass 6 completed **

C source file example.c is built.
C header file example.h is built.

These top level higher order procedures are not clonable (slow):
(nonkeyword_make-promise map-streams generate-vector runge-kutta-4)
These top level procedures create non-liftable closures (slow):
(nonkeyword_make-promise damped-oscillator map-streams scale-vector elementwise runge-

; Scheme (linux) script created by SLIB/batch Sun Apr  7 22:46:31 2002
; ================ Write file with C defines
(delete-file "scmflags.h")
```

```
(call-with-output-file
  "scmflags.h"
  (lambda (fp)
    (for-each
      (lambda (string) (write-line string fp))
      '("#define IMPLINIT \"Init5f3.scm\""
        "#define COMPILED_INITS init_example();"
        "#define CCLO"
        "#define FLOATS"))))
; ================ Compile C source files
(system "gcc -O2 -c continue.c scmmain.c findexec.c script.c time.c repl.c scl.c eval.
; ================ Link C object files
(system "gcc -rdynamic -o exscm continue.o scmmain.o findexec.o script.o time.o repl.o

Compilation finished at Sun Apr  7 22:46:44
```

Note Bene: '`#define CCLO`' must be present in `scmfig.h`.

In order to see calls to the C compiler and linker, do

```
(verbose 3)
```

before calling these functions.

2.2 Error Detection

Error detection during compilation is minimal. In case your scheme code is syntactically incorrect, hobbit may crash with no sensible error messages or it may produce incorrect C code.

Hobbit does not insert any type-checking code into the C output it produces. Eg, if a hobbit-compiled program applies '`car`' to a number, the program will probably crash with no sensible error messages.

Thus it is strongly suggested to compile only throughly debugged scheme code.

Alternatively, it is possible to compile all the primitives into calls to the SCM procedures doing type-checking. Hobbit will do this if you tell it to assume that all the primitives may be redefined. Put

```
(define compile-all-proc-redefined #t)
```

anywhere in top level of your scheme code to achieve this.

Note Bene: The compiled code using

```
(define compile-all-proc-redefined #t)
```

will typically be much slower than one produced without using

```
(define compile-all-proc-redefined #t).
```

All errors caught by hobbit will generate an error message

```
COMPILATION ERROR:
<description of the error>
```

and hobbit will immediately halt compilation.

2.3 Hobbit Options

1. Selecting the type of arithmetics.

   By default hobbit assumes that only immediate (ie small, up to 30 bits) integers are used. It will automatically assume general arithmetics in case it finds any non-immediate numbers like 1.2 or 10000000000000 or real-only procedures like `real-sin` anywhere in the source.

   Another way to make Hobbit assume that generic arithmetic supported by SCM (ie exact and/or inexact reals, bignums) is also used, is to put the following line somewhere in your scheme source file:

   ```
   (define compile-allnumbers t)
   ```

   where *t* is arbitrary.

   In that case all the arithmetic primitives in all the given source files will be assumed to be generic. This will make operations with immediate integers much slower. You can use the special immediate-integer-only forms of arithmetic procedures to recover:

   ```
   %negative?  %number?    %>      %>=     %=      %<=     %<
   %positive?  %zero?      %eqv?   %+      %-      %*      %/
   ```

   See Chapter 3 [The Language Compiled], page 8.

2. Redefinition of procedures.

   By default hobbit assumes that neither primitives nor compiled procedures are redefined, neither before the compiled program is initialized, during its work or later via the interpreter.

   Hobbit checks the compiled source and whenever some variable bar is defined as a procedure, but is later redefined, or `set!` is applied to bar, then hobbit assumes thas this particular variable bar is redefinable. bar may be a primitive (eg '`car`') or a name of a compiled procedure.

   Note Bene: According to the Report 4 it is **NOT** allowed to use scheme keywords as variables (you may redefine these as macros defined by defmacro, though):

   ```
   => and begin case cond define delay do else if lambda
   let let letrec or quasiquote quote set! unquote unquote-splicing
   ```

   If you want to be able to redefine some procedures, eg. '`+`' and '`baz`', then put both

   ```
   (set! + +)
   (set! baz baz)
   ```

   somewhere into your file.

   As a consequence hobbit will generate code for '`+`' and '`baz`' using the run-time values of these variables. This is generally much slower than using non-redefined '`+`' and '`baz`' (especially for '`+`').

   If you want to be able to redefine all the procedures, both primitives (eg '`car`') and the compiled procedures, then put the following into the compiled file:

   ```
   (define compile-all-proc-redefined t)
   ```

   where *t* is arbitrary.

   If you want to be able to redefine all the compiled procedures, but not the scheme primitives, then put the following into the compiled file:

   ```
   (define compile-new-proc-redefined t)
   ```

where t is arbitrary.

Again, remember that redefinable procedures will be typically much slower than non-redefinable procedures.

3. Inlined variables and procedures.

   You may inline top-level-defined variables and procedures. Notice that inlining is DIFFERENT for variables and procedures!

   NEVER inline variables or procedures which are `set!` or redefined anywhere in you program: this will produce wrong code.

   - You may declare certain top-level defined variables to be inlined. For example, if the following variable foo is declared to be inlined

     ```
     (define foo 100)
     ```

   then '`foo`' will be everywhere replaced by '`100`'.

   To declare some variables foo and bar to be inlined, put a following definition anywhere into your file:

   ```
   (define compile-inline-vars '(foo bar))
   ```

   Usually it makes sense to inline only these variables whose value is either a small integer, character or a boolean.

   Note Bene: Do not use this kind of inlining for inlining procedures! Use the following for procedures:

   - You may declare certain procedures to be inlined. For example, if the following foo is declared to be inlined

     ```
     (define (foo x) (+ x 2))
     ```

   then any call

   ```
   (foo something)
   ```

   will be replaced by

   ```
   (+ something 2)
   ```

   Inlining is **NOT** safe for variable clashes – in other words, it is not "hygienic".

   Inlining is **NOT** safe for recursive procedures – if the set of inlined procedures contains either immediate or mutual (foo calling bar, bar calling foo) recursion, the compiler will not terminate. To turn off full inlining (harmful for recursive funs), change the definition of the *full-inlining-flag* in the section "compiler options" to the value #f instead of #t.

   To declare some procedures foo and bar to be inlined, put a following definition anywhere into your file:

   ```
   (define compile-inline '(foo bar))
   ```

4. Speeding up vectors:

   Put

   ```
   (define compile-stable-vectors '(baz foo))
   ```

   into your file to declare that baz and foo are vector names defined once on the top level, and `set!` is never applied to them (`vector-set!` is, of course, allowed). This speeds up vector reference to those vectors by precomputing their location.

5. Speeding up and hiding certain global variables:

   Put

   ```
   (define compile-uninterned-variables '(bazvar foovar))
   ```

   into your file to declare that bazvar and foovar are defined on the top level and they do always have an immediate value, ie a boolean, immediate (30-bit) integer or a character. Then bazvar and foovar will **NOT** be accessible from the interpreter. They'll be compiled directly into static C vars and used without an extra C *-operation prefixed to other global scheme variables.

6. Intermediate files

   To see the output of compiler passes, change the following definition in `hobbit.scm`.

   ```
   (define *build-intermediate-files* #f)
   ```

   to:

   ```
   (define *build-intermediate-files* #t)
   ```

7. Name clashes

   It may happen that several originally different scheme variable names are represented by one and the same C variable. This will happen, for example, if you have separate variables a-1 and a_1.

   If such (or any other) name clashes occur you may need to change some control variables in the first sections of `hobbit.scm` (up to the section "global variable defs") or just rename some variables in your scheme program.

8. Other options

   See various control variables in the first sections of `hobbit.scm` (up to section "global variable defs").

2.4 CC Optimizations

When using the C compiler to compile the C code output by hobbit, always use strong optimizations (eg. '`cc -xO3`' for cc on Sun, '`gcc -O2`' or '`gcc -O3`' for gcc). Hobbit does not attempt to do optimizations of the kind we anticipate from the C compiler, therefore it often makes a serious difference whether the C compiler is run with a strong optimization flag or not.

For the final and fast version of your program you may want to first recompile the whole scm (scmlit for the version scm4e2) using the '`-DRECKLESS`' flag suppressing error checking: the hobbit-compiled code uses some SCM primitives in the compiled files with the suffix .o, and a number of these primitives become faster when error checking is disabled by '`-DRECKLESS`'. Notice that hobbit never inserts error checking into the code it produces.

3 The Language Compiled

Calls to `load` or `require` occurring at the top level of a file being compiled are ignored. Calls to `load` or `require` within a procedure are compiled to call (interpreted) `load` or `require` as appropriate.

Several SCM and SLIB extensions to the Scheme report are recognized by hobbit as Scheme primitives.

3.1 Macros

The Common-lisp style defmacro implemented in SCM is recognized and procedures defined by defmacro are expanded during compilation.

Note Bene: any macro used in a compiled file must be also defined in one of the compiled files.

'`#.<`*expression*`>`' is read as the object resulting from the evaluation of `<`*expression*`>`. The calculation is performed during compile time. Thus `<`*expression*`>` must not contain variables defined or `set!` in the compiled file.

3.2 SCM Primitive Procedures

Real-only versions of transcedental procedures (warning: these procedures are not compiled directly into the corresponding C library procedures, but a combination of internal SCM procedures, guaranteeing exact correspondence with the SCM interpreter while hindering the speed):

```
real-sqrt real-exp real-ln real-expt real-sin real-cos real-tan
real-asin real-acos real-atan real-sinh real-cosh real-tanh real-asinh
real-acosh real-atanh
```

Note Bene: These procedures are compiled to faster code than the corresponding generic versions `sqrt`, `abs`, `...` `expt`.

A selection of other extra primitives in SCM is also recognized as primitives. eg. `get-internal-run-time`, `quit`, `abort`, `restart`, `chdir`, `delete-file`, `rename-file`.

3.3 SLIB Logical Procedures

The following bitwise procedures in the scheme library file `logical.scm` are compiled directly to fast C operations on immediate integers (small 30-bit integers) (Scheme library funs in the upper row, C ops below):

```
logand logior logxor lognot logsleft logsright
   &      |     ^      ~       <<        >>
```

The following alternative names `logical:logand`, `logical:logior`, `logical:logxor`, `logical:lognot`, and `ash` are compiled for the generic case, not immediate-integers-only and are thus much slower.

Notice that the procedures `logsleft`, `logsright` are **NOT** in the the library file `logical.scm`: the universal procedure `ash` is instead. Procedures `ash`, `logcount`,

`integer-length`, `integer-expt`, `bit-extract`, `ipow-by-squaring`, in `logical.scm` are not primtives and they are all compiled into calls to interpreted code.

`logsleft` and `logsright` are defined for non-compiled use in the file `scmhob.scm` included in the SCM distribution.

3.4 Fast Integer Calculations

The following primitives are for immediate (30-bit) integer-only arithmetics. The are compiled directly into the corresponding C operations plus some bitshifts if necessary. They are good for speed in case the compiled program uses BOTH generic arithmetics (reals, bignums) and immediate (30-bit) integer arithmetics. These procedures are much faster than corresponding generic procedures taking also reals and bignums. There is no point in using these unless the program as a whole is compiled using generic arithmetics, since otherwise all the arithmetics procedures are compiled directly into corresponding C operations anyway.

Note Bene: These primitives are **NOT** defined in SCM or its libraries. For non-compiled use they are defined in the file `scmhob.scm` included in the SCM distribution.

```
%negative?  %number?    %>      %>=     %=      %<=     %<
%positive?  %zero?      %eqv?   %+      %-      %*      %/
```

3.5 Force and Delay

The nonessential procedure `force` and syntax `delay` are implemented exactly as suggested in the report 4. This implementation deviates internally from the implementation of `force` and `delay` in the SCM interpeter, thus it is incorrect to pass a promise created by `delay` in the compiled code to the `force` used by interpreter, and vice-versa for the promises created by the interpreter.

3.6 Suggestions for writing fast code

The following suggestions may help you to write well-optimizable and fast code for the hobbit-scm combination. Roughly speaking, the main points are:

- minimizing consing and creation of new vectors and strings in speed-critical parts,
- minimizing the use of generic (non-integer) arithmetics in speed-critical parts,
- minimizing the usage of procedures as first-class objects (very roughly speaking, explicit lambda-terms and call/cc) in speed-critical parts,
- using special options and fast-compiled primitives of the compiler.

Here come the details.

1. Immediate arithmetics (ie using small, up to 30 bits integers) is much faster than generic (reals and bignums) arithmetics. If you have to use generic arithmetic in your program, then try to use special immediate arithmetics operations `%=`, `%<=`, `%+`, `%*`, ... for speed-critical parts of the program whenever possible.

   Also, if you use bitwise logical operations, try to use the immediate-integer-only versions

   ```
   logand logior logxor lognot logsleft logsright
   ```

   and not `logical:logand` or `ash`, for example.

2. Due to its inner stack-based architecture, the generic (not escape-only) continuations are very slow in SCM. Thus they are also slow in compiled code. Try to avoid continuations (calls to the procedure call-with-current-continuation and calls to the continuations it produces) in speed-critical parts.
3. In speed-critical parts of your program try to avoid using procedures which are redefined or defined by `set!`:

   ```
   (set! bar +)
   (set! f (lambda (x) (if (zero? x) 1 (* x (f (- x 1))))))
   ```

   anywhere in the compiled program. Avoid using compiler flags (see Section 2.3 [Hobbit Options], page 5):

   ```
   (define compile-all-proc-redefined t)
   (define compile-new-proc-redefined t)
   ```

4. Do not use complicated higher-order procedures in speed-critical parts. By *complicated* we mean *not clonable*, where clonability is defined in the following way (*Note Bene:* the primitives '`map`' and '`for-each`' are considered clonable and do not inflict a speed penalty).

   A higher-order procedure (HOP for short) is defined as a procedure with some of its formal arguments occuring in the procedure body in a function position, that is, as a first element of a list. Such an argument is called a *higher-order argument*.

   A HOP '`bar`' is clonable iff it satisfies the following four conditions:

   1. '`bar`' is defined as

      ```
      (define bar (lambda ...))
      ```

      or

      ```
      (define (bar ...) ...)
      ```

      on top level and bar is not redefined anywhere.
   2. the name '`bar`' occurs inside the body of bar only in a function position and not inside an internal lambda-term.
   3. Let f be a higher-order argument of bar. Any occurrence of f in bar has one of the following two forms:
      - f occurs in a function position,
      - f is passed as an argument to bar and in the call it occurs in the same position as in the argument list.
   4. Let f be a higher-order argument of bar. f does not occur inside a lambda-term occurring in bar.

      Examples:

      If '`member-if`' is defined on top level and is not redefined anywhere, then '`member-if`' is a clonable HOP:

      ```
      (define (member-if fn lst)
         (if (fn (car lst))
             lst
             (member-if fn (cdr lst)) ))
      ```

      member-if-not is not a clonable HOP (fn occurs in a lambdaterm):

```
(define (member-if-not fn lst)
  (member (lambda (x) (not (fn x))) lst) )
```

show-f is not a clonable HOP (fn occurs in a non-function position in (display fn)):

```
(define (show-f fn x)
    (set! x (fn x))
    (display fn)
    x)
```

5. In speed-critical parts avoid using procedures which return procedures.

   Eg, a procedure

   ```
   (define plus
     (lambda (x)
        (lambda (y) (+ y x)) ))
   ```

   returns a procedure.

6. A generalisation of the previous case 5:

   In speed-critical parts avoid using lambda-terms except in non-`set!` function definitions like

   ```
   (define foo (lambda ...)),
   (let ((x 1) (f (lambda ...))) ...)
   (let* ((x 1) (f (lambda ...))) ...)
   (let name ((x 1) (f (lambda ...))) ...)
   (letrec ((f (lambda ...)) (g (lambda ...))) ...)
   ```

   or as arguments to clonable HOP-s or primitives `map` and `for-each`, like

   ```
   (let ((x 0)) (map (lambda (y) (set! x (+ 1 x)) (cons x y)) list))
   (member-if (lambda (x) (< x 0)) list)
   ```

   where member-if is a clonable HOP.

   Also, avoid using variables with a procedural value anywhere except in a function position (first element of a list) or as an argument to a clonable HOP, `map` or `for-each`.

   Lambda-terms conforming to the current point are said to be liftable.

   Examples:

   ```
   (define (bar x) (let ((f car)) (f (f x))))
   ```

   has '`car`' in a non-function and non-HOP-argument position in `(f car)`, thus it is slower than

   ```
   (define (bar x) (let ((f 1)) (car (car x))))
   ```

   Similarly,

   ```
   (define (bar y z w)
     (let ((f (lambda (x) (+ x y))))
        (set! w f)
        (cons (f (car z))
              (map f z) )))
   ```

   has '`f`' occurring in a non-function position in `(set! w f)`, thus the lambda-term `(lambda (x) (+ x y))` is not liftable and the upper '`bar`' is thus slower than the following equivalent '`bar`' with a liftable inner lambda-term:

```
(define (bar y z w)
  (let ((f (lambda (x) (+ x y))))
     (set! w 0)
     (cons (f (car z))
           (map f z) )))
```

Using a procedure bar defined as

```
(define bar (let ((x 1)) (lambda (y) (set! x y) (+ x y))))
```

is slower than using a procedure bar defined as

```
(define *bar-x* 1)
(define bar (lambda (y) (set! *bar-x* y) (+ *bar-x* y)))
```

since the former definition contains a non-liftable lambda-term.

7. Try to minimize the amount of consing in the speed-critical program fragments, that is, a number of applications of cons, list, `map`, quasiquote (`) and vector->list during the time program is running. '`cons`' (called also by '`list`', '`map`' and '`quasiquote`') is translated into a C call to an internal cons procedure of the SCM interpreter. Excessive consing also means that the garbage collection happens more often. Do `(verbose 3)` to see the amount of time used by garbage collection while your program is running.

   Try to minimize the amount of creating new vectors, strings and symbols in the speed-critical program frgaments, that is, a number of applications of `make-vector`, `vector`, `list->vector`, `make-string`, `string-append`, `*->string`, `string->symbol`. Creating such objects takes typically much more time than consing.
8. The Scheme iteration construction '`do`' is compiled directly into the C iteration construction '`for`'. We can expect that the C compiler has some knowledge about '`for`' in the optimization stage, thus it is probably faster to use '`do`' for iteration than non-mutual tailrecursion (which is recognized by hobbit as such and is compiled into a jump to a beginning of a procedure) and certainly much faster than non-tail-recursion or mutual tailrecursion (the latter is not recognized by hobbit as such).
9. Declare small nonrecursive programs which do not contain let-s or lambdaterms as being inlinable.

   Declare globally defined variables which are never `set!` or redefined and whose value is a small integer, character or a boolean, as being inlinable. See Section 2.3 [Hobbit Options], page 5.
10. If possible, declare vectors as being stable. See Section 2.3 [Hobbit Options], page 5. This gives a minor improvement in speed.
11. If possible, declare critical global vars as being uninterned. See Section 2.3 [Hobbit Options], page 5. This gives a minor improvement in speed. Declare the global variables which are never `set!` and have an (unchanged) numeric or boolean value as being inlined. See Section 2.3 [Hobbit Options], page 5.

In addition, take the following into account:

- When using the C compiler to compile the C code output by hobbit, always use strong optimizations (eg. '`cc -xO3`' for cc on Sun, '`gcc -O2`' or '`gcc -O3`' for gcc).

Hobbit does not attempt to do optimizations of the kind we anticipate from the C compiler, therefore it often makes a big difference if the C compiler is run with a strong optimization flag or not.

- hobbit does not give proper tailrecursion behaviour for mutual tailrecursion (foo calling bar, bar calling foo tailrecursively).

  Hobbit guarantees proper tailrecursive behaviour for non-mutual tailrecursion (foo calling foo tailrecursively), provided that foo is not redefined anywhere and that foo is not a local function which occurs also in a non-function and non-clonable-HOP-argument position (i.e. cases 3 and 6 above).

4 Performance of Compiled Code

4.1 Gain in Speed

The author has so far compiled and tested a number of large programs (theorem provers for various logics and hobbit itself).

The speedup for the provers was between 25 and 40 times for various provable formulas. Comparison was made between the provers being interpreted and compiled with '`gcc -O2 -DRECKLESS`' on Sparcstation ELC in both cases.

The provers were written with care to make the compiled version run fast. They do not perform excessive consing and they perform very little arithmetic.

According to experiments made by A. Jaffer, the compiled form of the example program `pi.scm` was approximately 11 times faster than the interpreted form.

As a comparison, his hand-coded C program for the same algorithm of computing pi was about 12 times faster than the interpreted form. `pi.scm` spends most of of its time in immediate arithmetics, `vector-ref` and `vector-set!`.

P. Kelloma"ki has reported a 20-fold speedup for his generic scheme debugger. T. Moore has reported a 16-fold speedup for a large gate-level IC optimizer.

Self-compilation speeds Hobbit up only ca 10 times.

However, there are examples where the code compiled by hobbit runs actually slower than the same code running under interpreter: this may happen in case the speed of the code relies on non-liftable closures and proper mutual tailrecursion. See for example the closure-intensive benchmark CPSTAK in the following table.

4.2 Benchmarks

We will present a table with the performance of three scheme systems on a number of benchmarks: interpreted SCM, byte-compiled VSCM and hobbit-compiled code. The upper 13 benchmarks of the table are the famous Gabriel benchmarks (originally written for lisp) modified for scheme by Will Clinger. The lower five benchmarks of the table are proposed by other people. *Selfcompile* is the self-compile time of Hobbit.

Hobbit performs well on most of the benchmarks except CPSTAK and CTAK: CPSTAK is a closure-intensive tailrecursive benchmark and CTAK is a continuations-intensive benchmark. Hobbit performs extremely well on these benchmarks which essentially satisfy the criterias for well-optimizable code outlined in the section 6 above.

FFT is real-arithmetic-intensive.

All times are in seconds.

SCM 4c0(U) and 1.1.5*(U) (the latter is the newest version of VSCM) are compiled and run by Matthias Blume on a DecStation 5000 (Ultrix). VSCM is a bytecode-compiler using continuation-passing style, and is well optimized for continuations and closures.

SCM 4e2(S) and Hobbit4b(S) compiled (with '`cc -xO3`') and run by Tanel Tammet on a Sun SS10 (lips.cs.chalmers.se). Hobbit is a Scheme-to-C compiler for SCM, the code it produces does not do any checking. SCM and hobbit are not optimized for continuations. Hobbit is not optimized for closures and proper mutual tailrecursion.

SCM and Hobbit benchmarks were run giving ca 8 MB of free heap space before each test.

Benchmark	SCM 4c0(U)	1.1.5*(U)	SCM 4e2(S)	Hobbit4b(S)
Deriv	3.40	3.86	2.9	0.18
Div-iter	3.45	2.12	2.6	0.083
Div-rec	3.45	2.55	3.5	0.42
TAK	1.81	1.71	1.4	0.018
TAKL	14.50	11.32	13.8(1.8 in gc)	0.13
TAKR	2.20	1.64	1.7 1.5	0.018
Destruct	?	?	7.4(1.8 in gc)	0.18
Boyer	?	?	27.(3.8 in gc)	1.9
CPSTAK	2.72	2.64	2.0 1.92	3.46(2.83 in gc)
CTAK	31.0	4.11	memory	memory
CTAK(7 6 1)	?	?	0.83	0.74
FFT	12.45	15.7	11.4 10.8	1.0
Puzzle	0.28	0.41	0.46(0.22 gc)	0.03
(recfib 25)	?	?	4.1	0.079
(recfib 30)	?	?	55. (10.in gc)	0.87
(pi 300 3)	?	?	7.4	0.46
(hanoi 15)	?	?	0.68	0.007
(hanoi 20)	?	?	31. (9. in gc)	0.2

4.3 Benchmark Sources

A selection of (smaller) benchmark sources

4.3.1 Destruct

```
;;;; Destructive operation benchmark
(define (destructive n m)
  (let ((l (do ((i 10 (- i 1))
                (a '() (cons '() a)))
               ((= i 0) a))))
    (do ((i n (- i 1)))
        ((= i 0))
      (if (null? (car l))
          (do ((l l (cdr l)))
              ((null? l))
            (or (car l) (set-car! l (cons '() '())))
            (append! (car l) (do ((j m (- j 1))
                                  (a '() (cons '() a)))
                                 ((= j 0) a))))
          (do ((l1 l (cdr l1))
               (l2 (cdr l) (cdr l2)))
              ((null? l2))
            (set-cdr! (do ((j (quotient (length (car l2)) 2) (- j 1))
                           (a (car l2) (cdr a)))
                          ((zero? j) a)
                        (set-car! a i))
                      (let ((n (quotient (length (car l1)) 2)))
                        (cond ((= n 0) (set-car! l1 '()) (car l1))
                              (else (do ((j n (- j 1))
                                         (a (car l1) (cdr a)))
                                        ((= j 1)
                                         (let ((x (cdr a)))
                                           (set-cdr! a '()) x))
                                      (set-car! a i)))))))))))
;; call:  (destructive 600 50)
```

4.3.2 Recfib

```
(define (recfib x)
  (if (< x 2)
      x
      (+ (recfib (- x 1))
         (recfib (- x 2)))))
```

4.3.3 div-iter and div-rec

```
;;;; Recursive and iterative benchmark divides by 2 using lists of ()'s.
(define (create-n n)
  (do ((n n (- n 1))
       (a '() (cons '() a)))
      ((= n 0) a)))
(define *ll* (create-n 200))
(define (iterative-div2 l)
  (do ((l l (cddr l))
       (a '() (cons (car l) a)))
      ((null? l) a)))
(define (recursive-div2 l)
  (cond ((null? l) '())
        (else (cons (car l) (recursive-div2 (cddr l))))))
(define (test-1 l)
  (do ((i 300 (- i 1))) ((= i 0))
    (iterative-div2 l)
    (iterative-div2 l)
    (iterative-div2 l)
    (iterative-div2 l)))
(define (test-2 l)
  (do ((i 300 (- i 1))) ((= i 0))
    (recursive-div2 l)
    (recursive-div2 l)
    (recursive-div2 l)
    (recursive-div2 l)))
;; for the iterative test call: (test-1 *ll*)
;; for the recursive test call: (test-2 *ll*)
```

4.3.4 Hanoi

```
;;; C optimiser should be able to remove the first recursive call to
;;; move-them.  But Solaris 2.4 cc, gcc 2.5.8, and hobbit don't.
(define (hanoi n)
  (letrec ((move-them
            (lambda (n from to helper)
              (if (> n 1)
                  (begin
                    (move-them (- n 1) from helper to)
                    (move-them (- n 1) helper to from))))))
    (move-them n 0 1 2)))
```

4.3.5 Tak

```
;;;; A vanilla version of the TAKeuchi function
(define (tak x y z)
  (if (not (< y x))
      z
      (tak (tak (- x 1) y z)
           (tak (- y 1) z x)
           (tak (- z 1) x y))))
;; call: (tak 18 12 6)
```

4.3.6 Ctak

```
;;;; A version of the TAK function that uses continuations
(define (ctak x y z)
  (call-with-current-continuation
   (lambda (k)
     (ctak-aux k x y z))))

(define (ctak-aux k x y z)
  (cond ((not (< y x)) (k z))
        (else (call-with-current-continuation
               (ctak-aux
                k
                (call-with-current-continuation
                 (lambda (k) (ctak-aux k (- x 1) y z)))
                (call-with-current-continuation
                 (lambda (k) (ctak-aux k (- y 1) z x)))
                (call-with-current-continuation
                 (lambda (k) (ctak-aux k (- z 1) x y))))))))

(define (id x) x)

(define (mb-test r x y z)
  (if (zero? r)
      (ctak x y z)
      (id (mb-test (- r 1) x y z))))
;;; call: (ctak 18 12 6)
```

4.3.7 Takl

```
;;;; The TAKeuchi function using lists as counters.
(define (listn n)
  (if (not (= 0 n))
      (cons n (listn (- n 1)))
      '()))

(define l18 (listn 18))
(define l12 (listn 12))
(define l6 (listn 6))

(define (mas x y z)
  (if (not (shorterp y x))
      z
      (mas (mas (cdr x) y z)
           (mas (cdr y) z x)
           (mas (cdr z) x y))))

(define (shorterp x y)
  (and (pair? y) (or (null? x) (shorterp (cdr x) (cdr y)))))
;; call: (mas l18 l12 l6)
```

4.3.8 Cpstak

```
;;;; A continuation-passing version of the TAK benchmark.
(define (cpstak x y z)
  (define (tak x y z k)
    (if (not (< y x))
        (k z)
        (tak (- x 1)
             y
             z
             (lambda (v1)
               (tak (- y 1)
                    z
                    x
                    (lambda (v2)
                      (tak (- z 1)
                           x
                           y
                           (lambda (v3)
                             (tak v1 v2 v3 k)))))))))
  (tak x y z (lambda (a) a)))
;;; call: (cpstak 18 12 6)
```

4.3.9 Pi

```
(define (pi n  . args)
      (let* ((d (car args))
             (r (do ((s 1 (* 10 s))
                     (i 0 (+ 1 i)))
                    ((>= i d) s)))
             (n (+ (quotient n d) 1))
             (m (quotient (* n d 3322) 1000))
             (a (make-vector (+ 1 m) 2)))
        (vector-set! a m 4)
        (do ((j 1 (+ 1 j))
             (q 0 0)
             (b 2 (remainder q r)))
            ((> j n))
          (do ((k m (- k 1)))
              ((zero? k))
            (set! q (+ q (* (vector-ref a k) r)))
            (let ((t (+ 1 (* 2 k))))
              (vector-set! a k (remainder q t))
              (set! q (* k (quotient q t)))))
          (let ((s (number->string (+ b (quotient q r)))))
            (do ((l (string-length s) (+ 1 l)))
                ((>= l d) (display s))
              (display #\0)))
          (if (zero? (modulo j 10)) (newline) (display #\space)))
        (newline)))
```

5 Principles of Compilation

5.1 Expansion and Analysis

1. Macros defined by defmacro and all the quasiquotes are expanded and compiled into equivalent form without macros and quasiquotes.

   For example, '(a , x) will be converted to (cons 'a (cons x '())).

2. Define-s with the nonessential syntax like

   ```
   (define (foo x) ...)
   ```

   are converted to `define`s with the essential syntax

   ```
   (define foo (lambda (x) ...))
   ```

   Non-top-level `define`s are converted into equivalent `letrec`-s.

3. Variables are renamed to avoid name clashes, so that any local variable may have a whole procedure as its scope. This renaming also converts let-s to let*-s. Variables which do not introduce potential name clashes are not renamed. For example,

   ```
   (define (foo x y)
      (let ((x y)
            (z x))
        (let* ((x (+ z x)))
          x)))
   ```

   is converted to

   ```
   (define foo
     (lambda (x y)
       (let* ((x__1 y)
              (z x)
              (x__2 (+ z x__1)))
         x__2)))
   ```

4. In case the set of procedures defined in one `letrec` is actually not wholly mutually recursive (eg, f1 calls f2, but f2 does not call f1, or there are three procedures, f1, f2, f3 so that f1 and f2 are mutually recursive but f3 is not called from f1 or f2 and it does not call them, etc), it is possible to minimize the number of additional variables passed to procedures.

   Thus `letrec`-s are split into ordered chunks using dependency analysis and topological sorting, to reduce the number of mutually passed variables. Wherever possible, `letrec`-s are replaced by `let*`-s inside these chunks.

5. Normalization is performed. This converts a majority of scheme control procedures like `cond`, `case`, `or`, `and` into equivalent terms using a small set of primitives. New variables may be introduced in this phase.

   In case a procedure like `or` or `and` occurs in the place where its value is treated as a boolean (eg. first argument of `if`), it is converted into an analogous boolean-returning procedure, which will finally be represented by an analogous C procedure (eg. `||` or &&).

   Associative procedures are converted into structures of corresponding nonassociative procedures. List is converted to a structure of cons-s.

Map and `for-each` with more than two arguments are converted into an equivalent do-cycle. `map`-s and `for-each`-s with two arguments are treated as if they were defined in the compiled file – the definitions `map1` and `for-each1` are automatically included, if needed.

There is an option in `hobbit.scm` to make all `map`-s and `for-each`-s be converted into equivalent do-loops, avoiding the use of `map1` and/or `for-each1` altogether.

6. Code is analysed for determining which primitive names and compiled procedure names are assumed to be redefinable.
7. Analysing HOP clonability: hobbit will find a list of clonable HOP-s with information about higher-order arguments.

   Criterias for HOP clonability are given in the section 6.4.
8. Analysis of liftability: hobbit will determine which lambda-terms have to be built as real closures (implemented as a vector where the first element is a pointer to a function and the rest contain values of environment variables or environment blocks of surrounding code) and which lambda-terms are liftable.

   Liftability analysis follows the criterias given in section 6.5 and 6.6.

5.2 Building Closures

Here Hobbit produces code for creating real closures for all the lambda-terms which are not marked as being liftable by the previous liftability analysis.

Global variables (eg x-glob) are translated as pointers (locations) to SCM objects and used via a fetch: *x_glob (or a fetch macro GLOBAL(x-glob) which translates to *x_glob).

While producing closures hobbit tries to minimize the indirection levels necessary. Generally a local variable x may have to be translated to an element of a vector of local variables built in the procedure creating x. If x occurs in a non-liftable closure, the whole vector of local variables is passed to a closure.

Such a translation using a local vector will only take place if either x is `set!` inside a non-liftable lambda-term or x is a name of a recursively defined non-liftable function, and the definition of x is irregular. The definition of x is irregular if x is given the non-liftable recursive value t by extra computation, eg as

```
(set! x (let ((u 1)) (lambda (y) (display u) (x (+ u 1)))))
```

and not as a simple lambdaterm:

```
(set! x (lambda (y) (display x) (x (+ y 1))))
```

In all the other cases a local scheme variable x is translated directly to a local C variable x having the type SCM (a 32-bit integer). If such an x occurs in a non-liftable closure, then only its value is passed to a closure via the closure-vector. In case the directly-translated variable x is passed to a liftable lambda-term where it is `set!`, then x is passed indirectly by using its address &x. In the lifted lambda-term it is then accessed via *.

If all the variables x1, ..., xn created in a procedure can be translated directly as C variables, then the procedure does not create a special vector for (a subset of) local variables.

An application (foo ...) is generally translated to C by an internal apply of the SCM interpreter: apply(GLOBAL(foo), ...). Using an internal apply is much slower than using direct a C function call, since:

- there is an extra fetch by GLOBAL(foo),
- internal apply performs some computations,
- the arguments of foo are passed as a list constructed during application: that is, there is a lot of expensive consing every time foo is applied via an internal apply.

However, in case foo is either a name of a non-redefined primitive or a name of a non-redefined liftable procedure, the application is translated to C directly without the extra layer of calling apply: foo(...).

Sometimes lambda-lifting generates the case that some variable x is accessed not directly, but by *x. See the next section.

Undefined procedures are assumed to be defined via interpreter and are called using an internal apply.

5.3 Lambda-lifting

When this pass starts, all the real (nonliftable) closures have been translated to closure-creating code. The remaining lambda-terms are all liftable.

Lambda-lifting is performed. That is, all procedures defined inside some other procedure (eg. in `letrec`) and unnamed lambda-terms are made top-level procedure definitions. Any N variables not bound in such procedures which were bound in the surrounding procedure are given as extra N first parameters of the procedure, and whenever the procedure is called, the values of these variables are given as extra N first arguments.

For example:

```
(define foo
  (lambda (x y)
     (letrec ((bar (lambda (u) (+ u x))))
        (bar y) )))
```

is converted to

```
(define foo
  (lambda (x y)
    (foo-fn1 x y) ))

(define foo-fn1
  (lambda (x u)
    (+ u x) ))
```

The case of mutually recursive definitions in `letrec` needs special treatment – all free variables in mutually recursive funs have, in general, to be passed to each of those funs. For example, in

```
(define (foo x y z i)
  (letrec ((f1 (lambda (u) (if x (+ (f2 u) 1))))
           (f2 (lambda (v) (if (zero? v) 1 (f1 z)))) )
    (f2 i) ))
```

the procedure f1 contains a free variable x and the procedure f2 contains a free variable z. Lambda-lifted f1 and f2 must each get both of these variables:

```
(define (foo x y z i)
   (foo-fn2 x z i) )

(define foo-fn1
  (lambda (x z u) (if x (+ (foo-fn2 x z u) 1))) )

(define foo-fn2
  (lambda (x z v) (if (zero? v) 1 (foo-fn1 x z z))) )
```

Recall that hobbit has already done dependency analysis and has split the original `letrec` into smaller chunks according to this analysis: see pass 1.

Whenever the value of some free variable is modified by `set!` in the procedure, this variable is passed by reference instead. This is not directly possible in scheme, but it is possible in C.

```
(define foo
  (lambda (x y z)
     (letrec ((bar (lambda (u) (set! z (+ u x z)))))
        (bar y)
        z)))
```

is converted to incorrect scheme:

```
(define foo
  (lambda (x y z)
    (foo-fn1 x (**c-adr** z) y)
    z))

(define foo-fn1
  (lambda (x (**c-adr** z) u)
    (set! (**c-fetch** z) (+ u x (**c-fetch** z))) ))
```

The last two will finally be compiled into correct C as:

```
SCM foo(x, y, z)
SCM x, y, z;
{
  foo_fn1(x, &z, y);
  return z;
}

SCM foo_fn1(x, z, u)
SCM x, u;
SCM *z;
{
  return (*z =  (u + x) + *z);
}
```

5.4 Statement-lifting

As the scheme do-construction is compiled into C for, but for cannot occur in all places in C (it is a statement), then if the do in a scheme procedure occurs in a place which will not be a statement in C, the whole do-term is lifted out into a new top-level procedure analogously to lambda-lifting. Any statement-lifted parts of some procedure foo are called foo_aux*n*, where *n* is a number.

The special C-ish procedure **return** is pushed into a scheme term as far as possible to extend the scope of statements in the resulting C program. For example,

```
(define foo
  (lambda (x y)
      (if x (+ 1 y) (+ 2 y)) ))
```

is converted to

```
(define foo
   (lambda (x y)
     (if x (**return** (+ 1 y)) (**return** (+ 2 y))) ))
```

Immediate tailrecursion (foo calling foo tailrecursively) is recognized and converted into an assignment of new values to args and a jump to the beginning of the procedure body.

5.5 Higher-order Arglists

All procedures taking a list argument are converted into ordinary non-list taking procedures and they are called with the list-making calls inserted. For example,

```
(define foo
  (lambda (x . y)
    (cons x (reverse y)) ))
```

is converted to

```
(define foo
  (lambda (x y)
    (cons x (reverse y)) ))
```

and any call to foo will make a list for a variable y. For example,

```
(foo 1 2 3)
```

is converted to

```
(foo 1 (cons 2 (cons 3 '()))).
```

All higher-order procedure calls where an argument-term contains unbound variables will generate a new instance (provided it has not been created already) of this higher-order procedure, carrying the right amount of free variables inside to right places.

For example, if there is a following definition:

```
(define (member-if fn lst)
   (if (fn (car lst))
       lst
       (member-if fn (cdr lst)) ))
```

and a call

```
(member-if (lambda (x) (eq? x y)) lst),
```

a new instance of member-if is created (if an analogous one has not been created before):

```
(define (member-if_inst1 tmp fn lst)
   (if (fn tmp (car lst))
       lst
       (member-if_inst1 tmp fn (cdr lst)) ))
```

and the call is converted to

```
(member-if_inst1 y foo lst)
```

and a top-level `define`

```
(define (foo y x) (eq? x y))
```

In addition, if the higher-order procedure is to be exported, an additional instance is created, which uses apply to call all argument-procedures, assuming they are defined via interpreter. The exportable higher-order procedure will have a name *fun*_exporthof, where *fun* is the name of the original procedure.

5.6 Typing and Constants

All C<->Scheme conversions for immediate objects like numbers, booleans and characters are introduced. Internal apply is used for undefined procedures. Some optimizations are performed to decrease the amount of C<->Scheme object conversions.

All vector, pair and string constants are replaced by new variables. These variables are instantiated to the right values by init_*foo**.

Procedures foo which are to be exported (made accesible to the interpreter), and which have an arity different from one of the following five templates: x, (), (x), (x y), (x y z), are made accessible via an additional procedure foo_wrapper taking a single list argument.

C Code Generation

More or less straightforward.

The type conversion between C objects and immediate Scheme objects of the type boolean, char and num is performed by macros. The scheme object `'()` is represented by the macro object EOL.

Intermediate files

Experiment yourself by defining:

```
(define *build-intermediate-files* #t)
```

instead of the default:

```
(define *build-intermediate-files* #f).
```

6 About Hobbit

6.1 The Aims of Developing Hobbit

1. Producing maximally fast C code from simple scheme code.

   By *simple* we mean code which does not rely on procedures returning procedures (closures) and nontrivial forms of higher-order procedures. All the latter are also compiled, but the optimizations specially target simple code fragments. Hobbit performs global optimization in order to locate such fragments.

2. Producing C code which would preserve as much original scheme code structure as possible, to enable using the output C code by a human programmer (eg. for introducing special optimizations possible in C). Also, this will hopefully help the C compiler to find better optimizations.

6.2 Manifest

`hobbit.scm`	the hobbit compiler.
`scmhob.scm`	the file defining some additional procedures recognized by hobbit as primitives. Use it with the interpreter only.
`scmhob.h`	the common headerfile for hobbit-compiled C files.
`hobbit.texi`	documentation for hobbit.

6.3 Author and Contributors

Tanel Tammet
Department of Computing Science
Chalmers University of Technology
University of Go"teborg
S-41296 Go"teborg Sweden

A. Jaffer (agj@alum.mit.edu), the author of SCM, has been of major help with a number of suggestions and hacks, especially concerning the interface between compiled code and the SCM interpreter.

Several people have helped with suggestions and detailed bug reports, e.g. David J. Fiander (davidf@mks.com), Gordon Oulsnam (STCS8004@IRUCCVAX.UCC.IE), Pertti Kelloma"ki (pk@cs.tut.fi), Dominique de Waleffe (ddw2@sunbim.be) Terry Moore (tmm@databook.com), Marshall Abrams (ab2r@midway.uchicago.edu). Georgy K. Bronnikov (goga@bronnikov.msk.su), Bernard Urban (Bernard.URBAN@meteo.fr), Charlie Xiaoli Huang, Tom Lord (lord@cygnus.com), NMICHAEL@us.oracle.com, Lee Iverson (leei@ai.sri.com), Burt Leavenworth (EDLSOFT@aol.com).

6.4 Future Improvements

1. Optimisations:
   - the calls to internal apply: we'd like to avoid the excessive consing of always building the list of arguments.
   - speeding up the creation of a vector for assignable closure-variables
   - several peephole optimisations.
2. Improve Variable creation and naming to avoid C function name clashes.
3. Report 4 macros.
4. Better error-checking.
5. Better liftability analysis.
6. More tailrecursion recognition.
7. Better numeric optimizations.
8. Fast real-only arithmetics: $eqv, $=, $>, $+, $*, etc.

6.5 Release History

[In February 2002, hobbit5x was integrated into the SCM distribution. Changes since then are recorded in `scm/ChangeLog`.]

hobbit4d:

- the incorrect translation of char>?, char-ci>?, char>=?, char-ci>=? string>?, string-ci>?, string-ci>=?, string>=? reported by Burt Leavenworth (EDLSOFT@aol.com) was fixed.
- the name clash bug for new variables new_varN occurring in non-liftable closures (reported by Lee Iverson (leei@ai.sri.com)) was fixed.
- the major COPYRIGHT change: differently from all the previous versions of Hobbit, hobbit4d is Free Software.

hobbit4c:

- a liftability-analysis bug for `for-each` and `map` reported by Lee Iverson (leei@ai.sri.com) has been fixed.
- The output C code does not contain the unnecessary ;-s on separate lines any more.

hobbit4b: The following bugs have been fixed:

- Erroneous treatment of [and] inside symbols, reported by A. Jaffer (agj@alum.mit.edu).
- A bug in the liftability analysis, reported by A. Jaffer (agj@alum.mit.edu).
- A bug occurring in case arguments are evaluated right-to-left, which happens with Hobbit compiled by gcc on GNU/Linux. Reported and patched by George K. Bronnikov (goga@bronnikov.msk.su)
- A closure-building bug sometimes leading to a serious loss of efficiency (liftability not recognized), reported by NMICHAEL@us.oracle.com.
- A bug in the liftability analysis (non-liftable lambda-term inside a liftable lambda-term) reported by Lee Iverson (leei@ai.sri.com)

hobbit4a: Several bugs found in version4x are fixed.

hobbit4x (not public):

- A major overhaul: Hobbit is now able to compile full scheme, not just the fast liftable-clonable fragment.

  The optimizations done by the earlier versions are preserved.
- Numerous bugs found in earlier versions have been fixed.

hobbit3d: bugs found in the version 3c are fixed.

hobbit3c:

- the form

  ```
  (define foo (let ((x1 <t1>) ... (xn <tn>)) (lambda ...)))
  ```

  is now supported for all terms <ti> except procedures defined in the compiled files.
- macros are partially supported by doing a preprocessing pass using the procedures pprint-filter-file and defmacro:expand* defined in slib.
- the file `scmhob.scm` defining hobbit-recognized nonstandard procedures is created.
- the documentation is improved (thanks go to Aubrey for suggestions).

hobbit3b:

- Aubrey fixed some problems with the version 3.
- It is now OK to define procedures "by name" on top level.
- It is now OK to apply "apply", etc to procedures defined in the compiled file. Compiled procedures may now be passed to procedures not defined but still called in the compiled files.

hobbit3:

- Generic arithmetic supported by SCM (exact and inexact reals, bignums) is made available.
- The #. special syntactic form of SCM is made available.
- Procedures with chars are compiled open-coded, making them faster.
- The bug concerning strings containing an embedded \nl char is corrected (thanks to Terry Moore, (tmm@databook.com)).
- The special declaration compile-stable-vectors for optimizing vector access is introduced.
- Source code may contain top-level computations, top-level loads are ignored.
- The bug causing "or" to (sometimes) lose tailrecursiveness is corrected.
- Hobbit now allows the following very special form:

  ```
  (define foo (let ((bar bar)) (lambda ...)))
  ```

  Notice `(bar bar)`. See the section 5 above. It will produce wrong code if bar is redefined.

  There were several versions of the 2-series, like 2.x, which were not made public. The changes introduced are present in the version 3.

hobbit2:

- The following bitwise procedures in the scheme library file `logical.scm` are compiled directly to C (Scheme library funs in the upper row, C ops below):

```
logand logior logxor lognot logsleft logsright
   &      |     ^      ~       <<        >>
```

  Notice that the procedures `logsleft`, `logsright` are **NOT** in the the library file `logical.scm`: the universal procedure `ash` is instead. Procedures `ash`, `logcount`, `integer-length`, `integer-expt`, `bit-extract` in `logical.scm` are not recognized by hobbit.

hobbit1a3 (not public):

- the `letrec`-sorting bug often resulting in not recognizing procedures defined in `letrec` (or local `define`s) has been corrected.
- the primitives `string` and `vector` are now compiled correctly.

hobbit1a2 (not public):

- any fixed arity procedure (including primitives) may be passed to any higher-order procedure by name. Variable arity procedures (eg primitives `list`, `+`, `display` and defined funs like `(define (foo x . y) x)`) must not be passed to new defined higher-order funs.
- some optimizations have been introduced for calls to `map` and `for-each`.
- (map list x y) bug has been corrected.
- Corrected self-compilation name clash between call_cc and call-cc.

hobbit1a1 (not public):

- named let is supported.
- the inlining bug is fixed: all procedures declared to be inlined are fully inlined, except when the flag *full-inlining-flag* is defined as #f.
- the `letrec` (or in-procedure define) bug where local procedure names were not recognized, is fixed.
- documentation says explicitly that definitions like

```
(define foo (let ((x 0)) (lambda (y) ...)))
```

  are assumed to be closure-returning procedures and are prohibited.
- documentation allows more liberty with passing procedures to higher-order funs by dropping the general requirement that only unnamed lambda-terms may be passed. Still, primitives and list-taking procedures may not be passed by name.
- documentation prohibits passing lambda-terms with free variables to recursive calls of higher-order procedures in the definition of a higher-order procedure.

hobbit1: the first release

Index