A Byte Code Compiler for R

Luke Tierney
Department of Statistics and Actuarial Science
University of Iowa

October 2, 2014

This document presents the current implementation of the byte code compiler for R. The compiler produces code for a virtual machine that is then executed by a virtual machine runtime system. The virtual machine is a stack based machine. Thus instructions for the virtual machine take arguments off a stack and may leave one or more results on the stack. Byte code objects consists of an integer vector representing instruction opcodes and operands, and a generic vector representing a constant pool. The compiler is implemented almost entirely in R, with just a few support routines in C to manage compiled code objects.

The virtual machine instruction set is designed to allow much of the interpreter internals to be re-used. In particular, for now the mechanism for calling functions of all types from compiled code remains the same as the function calling mechanism for interpreted code. There are opportunities for efficiency improvements through using a different mechanism for calls from compiled functions to compiled functions, or changing the mechanism for both interpreted and compiled code; this will be explored in future work.

The style used by the compiler for building up code objects is imperative: A code buffer object is created that contains buffers for the instruction stream and the constant pool. Instructions and constants are written to the buffer as the compiler processes an expression tree, and at the end a code object is constructed. A more functional design in which each compiler step returns a modified code object might be more elegant in principle, but it would be more difficult to make efficient.

A multi-pass compiler in which a first pass produces an intermediate representation, subsequent passes optimize the intermediate representation, and a final pass produces actual code would also be useful and might be able to produce better code. A future version of the compiler may use this approach. But for now to keep things simple a single pass is used.

1 The compiler interface

The compiler can be used either explicitly by calling certain functions to carry out compilations, or implicitly by enabling compilation to occur automatically at certain points.

1.1 Explicit compilation

The primary functions for explicit compilation are `compile`, `cmpfun`, and `cmpfile`.

The `compile` function compiles an expression and returns a byte code object, which can then be passed to `eval`. A simple example is

```
> library(compiler)
> compile(quote(1+3))
<bytecode: 0x25ba070>
> eval(compile(quote(1+3)))
[1] 4
```

A closure can be compiled using `cmpfun`. If the function `f` is defined as

```
f <- function(x) {
    s <- 0.0
    for (y in x)
        s <- s + y
    s
}
```

then a compiled version is produced by

```
fc <- cmpfun(f)
```

We can then compare the performance of the interpreted and compiled versions:

```
> x <- as.double(1 : 10000000)
> system.time(f(x))
   user  system elapsed
  6.470   0.010   6.483
> system.time(fc(x))
   user  system elapsed
  1.870   0.000   1.865
```

A source file can be compiled with `cmpfile`. For now, the resulting file has to then be loaded with `loadcmp`. In the future it may make sense to allow `source` to either load a pre-compiled file or to optionally compile while sourcing.

1.2 Implicit compilation

Implicit compilation can be used to compile packages as they are installed or for just-in-time (JIT) compilation of functions or expressions. The mechanism for enabling these is experimental and likely to change.

For now, compilation of packages requires the use of lazy loading and can be enabled either by calling `compilePKGS` with argument `TRUE` or by starting R with the environment variable `R_COMPILE_PKGS` set to a positive integer value. In a UNIX-like environment, for example, installing a package with

```
env R_COMPILE_PKGS=1 R CMD INSTALL foo.tar.gz
```

will compile the functions in the package as they are written to the lazy loading data base.

If R is installed from source then the base and required packages can be compiled on installation using

```
make bytecode
```

This does not require setting the `R_COMPILE_PKGS` environment variable.

JIT compilation can be enabled from within R by calling `enableJIT` with a non-negative integer argument or by starting R with the environment variable `R_ENABLE_JIT` set to a non-negative integer. The possible values of the argument to `enableJIT` and their meanings are

0 turn off JIT

1 compile closures before they are called the first time

2 same as 1, plus compile closures before duplicating (useful for packages that store closures in lists, like lattice)

3 same as 2, plus compile all `for()`, `while()`, and `repeat()` loops before executing.

R may initially be somewhat sluggish if JIT is enabled and base and recommended packages have not been pre-compiled as almost everything will initially need some compilation.

2 The basic compiler

This section presents the basic compiler for compiling R expressions to byte code objects.

2.1 The compiler top level

R expressions consist of function calls, variable references, and literal constants. To create a byte code object representing an R expression the compiler has to walk the expression tree and emit code for the different node types in encounters. The code emitted may depend on the environment in which the expression will be evaluated as well as various compiler option settings.

The simplest function in the top level compiler interface is the function `compile`. This function requires an expression argument and takes two optional arguments: an environment and a list of options. The default environment is the global environment.

3 ⟨`compile` *function* 3⟩≡

```
compile <- function(e, env = .GlobalEnv, options = NULL) {
    cenv <- makeCenv(env)
    cntxt <- make.toplevelContext(cenv, options)
    cntxt$env <- addCenvVars(cenv, findLocals(e, cntxt))
    genCode(e, cntxt)
}
```

The supplied environment is converted into a compilation environment data structure. This compilation environment and any options provided are then used to construct a compiler context. The function `genCode` is then used to generate a byte code object for the expression and the constructed compilation context.

Compilation environments are described in Section 5 and compiler contexts in Section 4. The `genCode` function is defined as

4a ⟨`genCode` *function* 4a⟩≡

```
genCode <- function(e, cntxt, gen = NULL) {
    cb <- make.codeBuf(e)
    if (is.null(gen))
        cmp(e, cb, cntxt)
    else
        gen(cb, cntxt)
    codeBufCode(cb)
}
```

`genCode` creates a code buffer, fills the code buffer, and then calls `codeBufCode` to extract and return the byte code object. In the most common case `genCode` uses the low level recursive compilation function `cmp`, described in Section 2.3, to generate the code. For added flexibility it can be given a generator function that emits code into the code buffer based on the provided context. This is used in Section 10 for ****.

2.2 Basic code buffer interface

Code buffers are used to accumulate the compiled code and related constant values. A code buffer `cb` is a list containing a number of closures used to manipulate the content of the code buffer. In this section two closures are used, `putconst` and `putcode`.

The closure `cb$putconst` is used to enter constants into the constant pool. It takes a single argument, an arbitrary R object to be entered into the constant pool, and returns an integer index into the pool. The `cb$putcode` closure takes an instruction opcode and any operands the opcode requires and emits them into the code buffer. The operands are typically constant pool indices or labels, to be introduced in Section 3.

As an example, the `GETVAR` instruction takes one operand, the index in the constant pool of a symbol. The opcode for this instruction is `GETVAR.OP`. The instruction retrieves the symbol from the constant pool, looks up its value in the current environment, and pushes the value on the stack. If `sym` is a variable with value a symbol, then code to enter the symbol in the constant pool and emit an instruction to get its value would be

4b ⟨*example of emitting a* `GETVAR` *instruction* 4b⟩≡

```
ci <- cb$putconst(sym)
cb$putcode(GETVAR.OP, ci)
```

The complete code buffer implementation is given in Section 3.

2.3 The recursive code generator

The function `cmp` is the basic code generation function. It recursively traverses the expression tree and emits code as it visits each node in the tree.

Before generating code for an expression the function `cmp` attempts to determine the value of the expression by constant folding using the function `constantFold`. If constant folding is successful then `contantFold` returns a named list containing a `value` element. Otherwise it returns `NULL`. If constant folding is successful, then the result is compiled as a constant. Otherwise, the standard code generation process is used.

In the interpreter there are four types of objects that are not treated as constants, i.e. as evaluating to themselves: function calls of type `"language"`, variable references of type `"symbol"`, promises, and byte code objects. Neither promises nor byte code objects should appear as literals in code so an error is signaled for those. The language, symbol, and constant cases are each handled by their own code generators.

5a ⟨*generate code for expression* `e` 5a⟩≡
```
if (typeof(e) == "language")
    cmpCall(e, cb, cntxt)
else if (typeof(e) == "symbol")
    cmpSym(e, cb, cntxt, missingOK)
else if (typeof(e) == "bytecode")
    cntxt$stop(gettext("cannot compile byte code literals in code"),
               cntxt)
else if (typeof(e) == "promise")
    cntxt$stop(gettext("cannot compile promise literals in code"),
               cntxt)
else
    cmpConst(e, cb, cntxt)
```

The function `cmp` is then defined as

5b ⟨`cmp` *function* 5b⟩≡
```
cmp <- function(e, cb, cntxt, missingOK = FALSE) {
    ce <- constantFold(e, cntxt)
    if (is.null(ce)) {
        ⟨generate code for expression e 5a⟩
    }
    else
        cmpConst(ce$value, cb, cntxt)
}
```

The call code generator `cmpCall` will recursively call `cmp`.

2.4 Compiling constant expressions

The constant code generator `cmpConst` is the simplest of the three generators. A simplified generator can be defined as

5c ⟨*simplified* `cmpConst` *function* 5c⟩≡
```
cmpConst <- function(val, cb, cntxt) {
    ci <- cb$putconst(val)
    cb$putcode(LDCONST.OP, ci)
    if (cntxt$tailcall) cb$putcode(RETURN.OP)
}
```

This function enters the constant in the constant pool using the closure `cb$putconst`. The value returned by this closure is an index for the constant in the constant pool. Then the code generator emits an instruction to load the constant at the specified constant pool index and push it onto the stack. If the expression appears in tail position then a `RETURN` instruction is emitted as well.

Certain constant values, such as `TRUE`, `FALSE`, and `NULL` appear very often in code. It may be useful to provide and use special instructions for loading these. The resulting code will have slightly smaller constant pools and may be a little faster, though the difference is likely to be small. A revised definition of `cmpConst` that makes use of instructions for loading these particular values is given by

6 ⟨`cmpConst` *function* 6⟩≡

```
cmpConst <- function(val, cb, cntxt) {
    if (identical(val, NULL))
        cb$putcode(LDNULL.OP)
    else if (identical(val, TRUE))
        cb$putcode(LDTRUE.OP)
    else if (identical(val, FALSE))
        cb$putcode(LDFALSE.OP)
    else {
        ci <- cb$putconst(val)
        cb$putcode(LDCONST.OP, ci)
    }
    if (cntxt$tailcall) cb$putcode(RETURN.OP)
}
```

It might be useful to handle other constants in a similar way, such as `NA` or small integer values; this may be done in the future.

Ideally the implementation should be able to mark the values in the constant pool of a byte code object as read-only by setting the `NAMED` field to 2, but experience in testing shows that there are several packages in the wild that assume that an expression `TRUE`, for example, appearing in code will result in a freshly allocated value that can be freely modified in `.C` calls. It would be good to educate users not to do this, but for now the implementation duplicates all values as they are retrieved from the constant pool.

2.5 Compiling variable references

The function `cmpSym` handles compilation of variable references. For standard variables this involves entering the symbol in the constant pool, emitting code to look up the value of the variable at the specified constant pool location in the current environment, and, if necessary, emitting a `RETURN` instruction.

In addition to standard variables there is the ellipsis variable `...` and the accessors `..1`, `..2`, and so on that need to be considered. The ellipsis variable can only appear as an argument in function calls, so `cmp`, like the interpreter `eval` itself, should not encounter it. The interpreter signals an error if it does encounter a `...` variable, and the compiler emits code that does the same at runtime. The compiler also emits a warning at compile time. Variables representing formal parameters may not have values provided in their calls, i.e. may have missing values. In some cases this should signal an error; in others the missing value can be passed on (for example in expressions of the form `x[]`). To support this, `cmpSym` takes an optional argument for allowing missing argument values.

7a ⟨cmpSym *function* 7a⟩≡
```
cmpSym <- function(sym, cb, cntxt, missingOK = FALSE) {
    if (sym == "...") {
        notifyWrongDotsUse("...", cntxt)
        cb$putcode(DOTSERR.OP)
    }
    else if (is.ddsym(sym)) {
        ⟨emit code for ..n variable references 7b⟩
    }
    else {
        ⟨emit code for standard variable references 7c⟩
    }
}
```

References to `..n` variables are also only appropriate when a `...` variable is available, so a warning is given if that is not the case. The virtual machine provides instructions `DDVAL` and `DDVAL_MISSOK` for the case where missing arguments are not allowed and for the case where they are, and the appropriate instruction is used based on the `missingOK` argument to `cmpSym`.

7b ⟨*emit code for* `..n` *variable references* 7b⟩≡
```
if (! findLocVar("...", cntxt))
    notifyWrongDotsUse(sym, cntxt)
ci <- cb$putconst(sym)
if (missingOK)
    cb$putcode(DDVAL_MISSOK.OP, ci)
else
    cb$putcode(DDVAL.OP, ci)
if (cntxt$tailcall) cb$putcode(RETURN.OP)
```

There are also two instructions available for obtaining the value of a general variable from the current environment, one that allows missing values and one that does not.

7c ⟨*emit code for standard variable references* 7c⟩≡
```
if (! findVar(sym, cntxt))
    notifyUndefVar(sym, cntxt)
ci <- cb$putconst(sym)
if (missingOK)
    cb$putcode(GETVAR_MISSOK.OP, ci)
else
    cb$putcode(GETVAR.OP, ci)
if (cntxt$tailcall) cb$putcode(RETURN.OP)
```

For now, these instructions only take an index in the constant pool for the symbol as operands, not any information about where the variable can be found within the environment. This approach to obtaining the value of variables requires a search of the current environment for every variable reference. In a less dynamic language it would be possible to compute locations of variable bindings within an environment at compile time and to choose environment representations that allow constant time access to any variable's value. Since bindings in R can be added or removed at runtime this would require a semantic change that would need some form of declaration to make legitimate.

Another approach that may be worth exploring is some sort of caching mechanism in which the location of each variable is stored when it is first found by a full search, and that cached location is used until an event occurs that forces flushing of the cache. If such events are rare, as they typically are, then this may be effective.

2.6 Compiling function calls

Conceptually, the R function calling mechanism uses lazy evaluation of arguments. Thus calling a function involves three steps:

- finding the function to call
- packaging up the argument expressions into deferred evaluation objects, or promises
- executing the call

Code for this process is generated by the function `cmpCall`. A simplified version is defined as

8a ⟨*simplified* `cmpCall` *function* 8a⟩≡
```
cmpCall <- function(call, cb, cntxt) {
    cntxt <- make.callContext(cntxt, call)
    fun <- call[[1]]
    args <- call[-1]
    if (typeof(fun) == "symbol")
        cmpCallSymFun(fun, args, call, cb, cntxt)
    else
        cmpCallExprFun(fun, args, call, cb, cntxt)
}
```

Call expressions in which the function is represented by a symbol are compiled by `cmpCallSymFun`. This function emits a `GETFUN` instruction and then compiles the arguments.

8b ⟨`cmpCallSymFun` *function* 8b⟩≡
```
cmpCallSymFun <- function(fun, args, call, cb, cntxt) {
    ci <- cb$putconst(fun)
    cb$putcode(GETFUN.OP, ci)
    ⟨compile arguments and emit CALL instruction 8c⟩
}
```

The `GETFUN` instruction takes a constant pool index of the symbol as an operand, looks for a function binding to the symbol in the current environment, places it on the stack, and prepares the stack for handling function call arguments.

Argument compilation is carried out by the function `cmpCallArgs`, presented in Section 2.7, and is followed by emitting code to execute the call and, if necessary, return a result.

8c ⟨*compile arguments and emit* `CALL` *instruction* 8c⟩≡
```
cmpCallArgs(args, cb, cntxt)
ci <- cb$putconst(call)
cb$putcode(CALL.OP, ci)
if (cntxt$tailcall) cb$putcode(RETURN.OP)
```

The call expression itself is stored in the constant pool and is available to the CALL instruction.

Calls in which the function is represented by an expression other than a symbol are handled by cmpCallExprFun. This emits code to evaluate the expression, leaving the value in the stack, and then emits a CHECKFUN instruction. This instruction checks that the value on top of the stack is a function and prepares the stack for receiving call arguments. Generation of argument code and the CALL instruction are handled as for symbol function calls.

9a ⟨cmpCallExprFun *function* 9a⟩≡
```
cmpCallExprFun <- function(fun, args, call, cb, cntxt) {
    ncntxt <- make.nonTailCallContext(cntxt)
    cmp(fun, cb, ncntxt)
    cb$putcode(CHECKFUN.OP)
    ⟨compile arguments and emit CALL instruction 8c⟩
}
```

The actual definition of cmpCall is a bit more complex than the simplified one given above:

9b ⟨cmpCall *function* 9b⟩≡
```
cmpCall <- function(call, cb, cntxt) {
    cntxt <- make.callContext(cntxt, call)
    fun <- call[[1]]
    args <- call[-1]
    if (typeof(fun) == "symbol") {
        if (! tryInline(call, cb, cntxt)) {
            ⟨check the call to a symbol function 9c⟩
            cmpCallSymFun(fun, args, call, cb, cntxt)
        }
    }
    else {
        ⟨hack for handling break() and next() expressions 10b⟩
        cmpCallExprFun(fun, args, call, cb, cntxt)
    }
}
```

The main addition is the use of a tryInline function which tries to generate more efficient code for particular functions. This function returns TRUE if it has handled code generation and FALSE if it has not. Code will be generated by the inline mechanism if inline handlers for the particular function are available and the optimization level permits their use. Details of the inlining mechanism are given in Section 6.

In addition to the inlining mechanism, some checking of the call is carried out for symbol calls. The checking code is

9c ⟨*check the call to a symbol function* 9c⟩≡
```
if (findLocVar(fun, cntxt))
    notifyLocalFun(fun, cntxt)
else {
    def <- findFunDef(fun, cntxt)
    if (is.null(def))
        notifyUndefFun(fun, cntxt)
    else
```

```
        checkCall(def, call, function(w) notifyBadCall(w, cntxt))
}
```

and `checkCall` is defined as

10a ⟨`checkCall` *function* 10a⟩≡

```
## **** clean up to use tryCatch
## **** figure out how to handler multi-line deparses
## ****     e.g. checkCall(‘{‘, quote({}))
## **** better design would capture error object, wrap it up, and pass it on
checkCall <- function(def, call, signal = warning) {
    if (typeof(def) %in% c("builtin", "special"))
        def <- args(def)
    if (typeof(def) != "closure" || any.dots(call))
        NA
    else {
        old <-options()$show.error.messages
        if (is.null(old)) old <- TRUE
        options(show.error.messages=FALSE)
        msg <- try({match.call(def, call); NULL})
        options(show.error.messages=old)
        if (! is.null(msg)) {
            msg <- sub("\n$", "", sub("^E.*: ", "", msg))
            emsg <- gettextf("possible error in '%s': %s",
                             deparse(call, 20)[1], msg)
            if (! is.null(signal)) signal(emsg)
            FALSE
        }
        else TRUE
    }
}
```

Finally, for calls where the function is an expression a hack is currently needed for dealing with the way the parser currently parses expressions of the form `break()` and `next()`. To be able to compile as many `break` and `next` calls as possible as simple `GOTO` instructions these need to be handled specially to avoid placing things on the stack. A better solution would probably be to modify the parser to make expressions of the form `break()` be syntax errors.

10b ⟨*hack for handling* `break()` *and* `next()` *expressions* 10b⟩≡

```
## **** this hack is needed for now because of the way the
## **** parser handles break() and next() calls
if (typeof(fun) == "language" && typeof(fun[[1]]) == "symbol" &&
    as.character(fun[[1]]) %in% c("break", "next"))
    return(cmp(fun, cb, cntxt))
```

2.7 Compiling call arguments

Function calls can contain four kinds of arguments:

- missing arguments

- ... arguments
- general expressions

In the first and third cases the arguments can also be named. The argument compilation function `cmpCallArgs` loops over the argument lists and handles each of the three cases, in addition to signaling errors for arguments that are literal bytecode or promise objects:

11a ⟨`cmpCallArgs` *function* 11a⟩≡
```
cmpCallArgs <- function(args, cb, cntxt) {
    names <- names(args)
    pcntxt <- make.promiseContext(cntxt)
    for (i in seq_along(args)) {
        a <- args[[i]]
        n <- names[[i]]
        ⟨compile missing argument 11b⟩
        ⟨compile ... argument 11c⟩
        ⟨signal an error for promise or bytecode argument 11d⟩
        ⟨compile a general argument 12a⟩
    }
}
```

The missing argument case is handled by

11b ⟨*compile missing argument* 11b⟩≡
```
if (missing(a)) { ## better test for missing??
    cb$putcode(DOMISSING.OP)
    cmpTag(n, cb)
}
```

Computations on the language related to missing arguments are tricky. The use of `missing` is a little odd, but for now at least it does work.

An ellipsis argument ... is handled by the `DODOTS` instruction:

11c ⟨*compile ... argument* 11c⟩≡
```
else if (is.symbol(a) && a == "...") {
    if (! findLocVar("...", cntxt))
        notifyWrongDotsUse("...", cntxt)
    cb$putcode(DODOTS.OP)
}
```

A warning is issued if no ... argument is visible.

As in `cmp`, errors are signaled for literal bytecode or promise values as arguments.

11d ⟨*signal an error for promise or bytecode argument* 11d⟩≡
```
else if (typeof(a) == "bytecode")
    cntxt$stop(gettext("cannot compile byte code literals in code"),
               cntxt)
else if (typeof(a) == "promise")
    cntxt$stop(gettext("cannot compile promise literals in code"),
               cntxt)
```

A general non-constant argument expression is compiled to a separate byte code object which is stored in the constant pool. The compiler then emits a `MAKEPROM` instruction that uses the stored code object. Promises are not needed for literal constant arguments as these are self-evaluating. Within the current implementation both the evaluation process and use of `substitute` will work properly if constants are placed directly in the argument list rather than being wrapped in promises. This could also be done in the interpreter, though the benefit is less clear as a runtime determination of whether an argument is a constant would be needed. This may still be cheap enough compared to the cost of allocating a promise to be worth doing. Constant folding in `cmp` may also produce more constants, but promises are needed in this case in order for `substitute` to work properly. These promises could be created as evaluated promises, though it is not clean how much this would gain.

12a ⟨*compile a general argument* 12a⟩≡
```
else {
    if (is.symbol(a) || typeof(a) == "language") {
        ci <- cb$putconst(genCode(a, pcntxt))
        cb$putcode(MAKEPROM.OP, ci)
    }
    else
        cmpConstArg(a, cb, cntxt)
    cmpTag(n, cb)
}
```

For calls to closures the `MAKEPROM` instruction retrieves the code object, creates a promise from the code object and the current environment, and pushes the promise on the argument stack. For calls to functions of type `BULTIN` the `MAKEPROM` instruction actually executes the code object in the current environment and pushes the resulting value on the stack. For calls to functions of type `SPECIAL` the `MAKEPROM` instruction does nothing as these calls use only the call expression.

Constant arguments are compiled by `cmpConstArg`. Again there are special instructions for the common special constants `NULL`, `TRUE`, and `FALSE`.

12b ⟨`cmpConstArg` 12b⟩≡
```
cmpConstArg <- function(a, cb, cntxt) {
    if (identical(a, NULL))
        cb$putcode(PUSHNULLARG.OP)
    else if (identical(a, TRUE))
        cb$putcode(PUSHTRUEARG.OP)
    else if (identical(a, FALSE))
        cb$putcode(PUSHFALSEARG.OP)
    else {
        ci <- cb$putconst(a)
        cb$putcode(PUSHCONSTARG.OP, ci)
    }
}
```

Code to install names for named arguments is generated by `cmpTag`:

12c ⟨`cmpTag` *function* 12c⟩≡
```
cmpTag <- function(n, cb) {
    if (! is.null(n) && n != "") {
```

```
        ci <- cb$putconst(as.name(n))
        cb$putcode(SETTAG.OP, ci)
    }
}
```

The current implementation allocates a linked list of call arguments, stores tags in the list cells, and allocates promises. Alternative implementations that avoid some or all allocation are worth exploring. Also worth exploring is having an instruction specifically for calls that do not require matching of named arguments to formal arguments, since cases that use only order of arguments, not names, are quite common and are known at compile time. In the case of calls to functions with definitions known at compile time matching of named arguments to formal ones could also be done at compile time.

2.8 Discussion

The framework presented in this section, together with some support functions, is actually able to compile any legal R code. But this is somewhat deceptive. The R implementation, and the `CALL` opcode, support three kinds of functions: closures (i.e. R-level functions), primitive functions of type `BUILTIN`, and primitive functions of type `SPECIAL`. Primitives of type `BUILTIN` always evaluate their arguments in order, so creating promises is not necessary and in fact the `MAKEPROM` instruction does not do so — if the function to be called is a `BUILTIN` then `MAKEPROM` runs the code for computing the argument in the current environment and pushes the value on the stack. On the other hand, primitive functions of type `SPECIAL` use the call expression and evaluate bits of it as needed. As a result, they will be running interpreted code. Since core functions like the sequencing function `{` and the conditional evaluation function `if` are of type `SPECIAL` this means most non-trivial code will be run by the standard interpreter. This will be addressed by defining inlining rules that allow functions like `{` and `if` to be compiled properly.

3 The code buffer

The code buffer is a collection of closures that accumulate code and constants in variables in their defining environment. For a code buffer `cb` the closures `cb$putcode` and `cb$putconst` write an instruction sequence and a constant, respectively, into the code buffer. The closures `cb$code` and `cb$consts` extract the code vector and the constant pool.

The function `make.codeBuf` creates a set of closures for managing the instruction stream buffer and the constant pool buffer and returns a list of these closures for use by the compilation functions. In addition, the expression to be compiled into the code buffer is stored as the first constant in the constant pool; this can be used to retrieve the source code for a compiled expression.

13 ⟨`make.codeBuf` *function* 13⟩≡

```
make.codeBuf <- function(expr) {
    ⟨instruction stream buffer implementation 14a⟩
    ⟨constant pool buffer implementation 14b⟩
    ⟨label management interface 15a⟩
    cb <- list(code = getcode,
```

```
                const = getconst,
                putcode = putcode,
                putconst = putconst,
                makelabel = makelabel,
                putlabel = putlabel,
                patchlabels = patchlabels)
    cb$putconst(expr) ## insert expression as first constant.
    cb
}
```

The instruction stream buffer uses a list structure and a count of elements in use, and doubles the size of the list to make room for new code when necessary. By convention the first entry is a byte code version number; if the interpreter sees a byte code version number it cannot handle then it falls back to interpreting the uncompiled expression. The doubling strategy is needed to avoid quadratic compilation times for large instruction streams.

14a ⟨*instruction stream buffer implementation* 14a⟩≡

```
codeBuf <- list(.Internal(bcVersion()))
codeCount <- 1
putcode <- function(...) {
    new <- list(...)
    newLen <- length(new)
    while (codeCount + newLen > length(codeBuf))
        codeBuf <<- c(codeBuf, vector("list", length(codeBuf)))
    codeBuf[(codeCount + 1) : (codeCount + newLen)] <<- new
    codeCount <<- codeCount + newLen
}
getcode <- function() as.integer(codeBuf[1 : codeCount])
```

The constant pool is accumulated into a list buffer. The zero-based index of the constant in the pool is returned by the insertion function. Values are only entered once; if a value is already in the pool, as determined by `identical`, its existing index is returned. Again a size-doubling strategy is used for the buffer. `.Internal` functions are used both for performance reasons and to prevent duplication of the constants.

14b ⟨*constant pool buffer implementation* 14b⟩≡

```
constBuf <- vector("list", 1)
constCount <- 0
putconst <- function(x) {
    if (constCount == length(constBuf))
        constBuf <<- .Internal(growconst(constBuf))
    i <- .Internal(putconst(constBuf, constCount, x))
    if (i == constCount)
        constCount <<- constCount + 1
    i
}
getconst <- function()
    .Internal(getconst(constBuf, constCount))
```

Labels are used for identifying targets for branching instruction. The label management interface creates new labels with `makelabel` as character strings that are unique within the buffer.

These labels can then be included as operands in branching instructions. The `putlabel` function records the current code position as the value of the label.

15a ⟨*label management interface* 15a⟩≡
```
idx <- 0
labels <- vector("list")
makelabel <- function() { idx <<- idx + 1; paste0("L", idx) }
putlabel <- function(name) labels[[name]] <<- codeCount
```

Once code generation is complete the symbolic labels in the code stream need to be converted to numerical offset values. This is done by `patchlabels`. Labels can appear directly in the instruction stream os in lists that have been placed in the instruction stream; this is used for the `SWITCH` instruction.

15b ⟨*label management interface* 15a⟩+≡
```
patchlabels <- function() {
    offset <- function(lbl) {
        if (is.null(labels[[lbl]]))
            stop(gettextf("no offset recorded for label \"%s\"", lbl),
                 domain = NA)
        labels[[lbl]]
    }
    for (i in 1 : codeCount) {
        v <- codeBuf[[i]]
        if (is.character(v))
            codeBuf[[i]] <<- offset(v)
        else if (typeof(v) == "list") {
            off <- as.integer(lapply(v, offset))
            ci <- putconst(off)
            codeBuf[[i]] <<- ci
        }
    }
}
```

The contents of the code buffer is extracted into a code object by calling `codeBufCode`:

15c ⟨`codeBufCode` *function* 15c⟩≡
```
codeBufCode <- function(cb) {
    cb$patchlabels()
    .Internal(mkCode(cb$code(), cb$const()))
}
```

4 Compiler contexts

The compiler context object `cntxt` carries along information about whether the expression appears in tail position and should be followed by a return or, whether the result is ignored, or whether the expression is contained in a loop. The context object also contains current compiler option settings as well as functions used to issue warnings or signal errors.

4.1 Top level contexts

Top level compiler functions start by creating a top level context. The constructor for top level contexts takes as arguments the current compilation environment, described in Section 5, and a list of option values used to override default option settings. Top level expressions are assumed to be in tail position, so the `tailcall` field is initialized as `TRUE`. The `needRETURNJMP` specifies whether a call to the `return` function can use the `RETURN` instruction or has to use a `longjmp` via the `RETURNJMP` instruction. Initially using a simple `RETURN` is safe; this is set set to `TRUE` when compiling promises ad certain loops where `RETURNJMP` is needed.

16a ⟨`make.toplevelContext` *function* 16a⟩≡
```
make.toplevelContext <- function(cenv, options = NULL)
    structure(list(tailcall = TRUE,
                   needRETURNJMP = FALSE,
                   env = cenv,
                   optimize = getCompilerOption("optimize", options),
                   suppressAll = getCompilerOption("suppressAll", options),
                   suppressUndefined = getCompilerOption("suppressUndefined",
                                                         options),
                   call = NULL,
                   stop = function(msg, cntxt)
                       stop(simpleError(msg, cntxt$call)),
                   warn = function(x, cntxt) cat(paste("Note:", x, "\n"))),
              class = "compiler_context")
```

Errors are signaled using a version of `stop` that uses the current call in the compilation context. The default would be to use the call in the compiler code where the error was raised, and that would not be meaningful to the end user. Ideally `warn` should do something similar and also use the condition system, but for now it just prints a simple message to standard output.

4.2 Other compiler contexts

The `cmpCall` function creates a new context for each call it compiles. The new context is the current context with the `call` entry replaced by the current call — this is be useful for issuing meaningful warning and error messages.

16b ⟨`make.callContext` *function* 16b⟩≡
```
make.callContext <- function(cntxt, call) {
    cntxt$call <- call
    cntxt
}
```

Non-tail-call contexts are used when a value is being computed for use in a subsequent computation. The constructor returns a new context that is the current context with the tailcall field set to `FALSE`.

16c ⟨`make.nonTailCallContext` *function* 16c⟩≡
```
make.nonTailCallContext <- function(cntxt) {
    cntxt$tailcall <- FALSE
    cntxt
}
```

A no value context is used in cases where the computed value will be ignored. For now this is identical to a non-tail-call context, but it may eventually be useful to distinguish the two situations. This is used mainly for expressions other than the final one in `{` calls and for compiling the bodies of loops.

17a ⟨`make.noValueContext` *function* 17a⟩≡
```
make.noValueContext <- function(cntxt) {
    cntxt$tailcall <- FALSE
    cntxt
}
```

The compiler context for compiling a function is a new toplevel context using the function environment and the current compiler options settings.

17b ⟨`make.functionContext` *function* 17b⟩≡
```
make.functionContext <- function(cntxt, forms, body) {
    nenv <- funEnv(forms, body, cntxt)
    ncntxt <- make.toplevelContext(nenv)
    ncntxt$optimize <- cntxt$optimize
    ncntxt$suppressAll <- cntxt$suppressAll
    ncntxt$suppressUndefined <- cntxt$suppressUndefined
    ncntxt
}
```

The context for compiling the body of a loop is a no value context with the loop information available.

17c ⟨`make.loopContext` *function* 17c⟩≡
```
make.loopContext <- function(cntxt, loop.label, end.label) {
    ncntxt <- make.noValueContext(cntxt)
    ncntxt$loop <- list(loop = loop.label, end = end.label, gotoOK = TRUE)
    ncntxt
}
```

The initial loop context allows `break` and `next` calls to be implemented as `GOTO` instructions. This is OK for calls that are in top level position relative to the loop. Calls that occur in promises or in other contexts where the stack has changed from the loop top level state need stack unwinding and cannot be implemented as `GOTO` instructions. These should should be compiled with contexts that have the `loop$gotoOK` field set to `FALSE`. The promise context does this for promises and the argument context for other settings. The promise context also sets `needRETURNJMP` to `TRUE` since a `return` call that is triggered by forcing a promise requires a `longjmp` to return from the appropriate function.

17d ⟨`make.argContext` *function* 17d⟩≡
```
make.argContext <- function(cntxt) {
    cntxt$tailcall <- FALSE
    if (! is.null(cntxt$loop))
        cntxt$loop$gotoOK <- FALSE
    cntxt
}
```

18a ⟨make.promiseContext *function* 18a⟩≡
```
make.promiseContext <- function(cntxt) {
    cntxt$tailcall <- TRUE
    cntxt$needRETURNJMP <- TRUE
    if (! is.null(cntxt$loop))
        cntxt$loop$gotoOK <- FALSE
    cntxt
}
```

4.3 Compiler options

Default compiler options are maintained in an environment. For now, the supported options are `optimize`, which is initialized to level 2, and two options for controlling compiler messages. The `suppressAll` option, if `TRUE`, suppresses all notifications. The `suppressUndefined` option can be `TRUE` to suppress all notifications about undefined variables and functions, or it can be a character vector of the names of variables for which warnings should be suppressed.

18b ⟨*compiler options data base* 18b⟩≡
```
compilerOptions <- new.env(hash = TRUE, parent = emptyenv())
compilerOptions$optimize <- 2
compilerOptions$suppressAll <- FALSE
compilerOptions$suppressUndefined <-
    c(".Generic", ".Method", ".Random.seed", ".self")
```

Options are retrieved with the `getCompilerOption` function.

18c ⟨getCompilerOption *function* 18c⟩≡
```
getCompilerOption <- function(name, options = NULL) {
    if (name %in% names(options))
        options[[name]]
    else
        get(name, compilerOptions)
}
```

The `suppressAll` function determines whether a context has its `supressAll` property set to `TRUE`.

18d ⟨suppressAll *function* 18d⟩≡
```
suppressAll <- function(cntxt)
    identical(cntxt$suppressAll, TRUE)
```

The `suppressUndef` function determines whether undefined variable or function definition notifications for a particular variable should be suppressed in a particular compiler context.

18e ⟨suppressUndef *function* 18e⟩≡
```
suppressUndef <- function(name, cntxt) {
    if (identical(cntxt$suppressAll, TRUE))
        TRUE
    else {
        suppress <- cntxt$suppressUndefined
        if (is.null(suppress))
            FALSE
```

```
        else if (identical(suppress, TRUE))
            TRUE
        else if (is.character(suppress) && as.character(name) %in% suppress)
            TRUE
        else FALSE
    }
}
```

At some point we will need mechanisms for setting default options from the interpreter and in package meta-data. A declaration mechanism for adjusting option settings locally will also be needed.

4.4 Compiler notifications

Compiler notifications are currently sent by calling the context's `warn` function, which in turn prints a message to standard output. It would be better to use an approach based on the condition system, and this will be done eventually. The use of separate notification functions for each type of issue signaled is a step in this direction.

Undefined function and undefined variable notifications are issued by `notifyUndefFun` and `notifyUndefVar`. These both use `suppressUndef` to determine whether the notification should be suppressed in the current context.

19a ⟨`notifyUndefFun` *function* 19a⟩≡

```
notifyUndefFun <- function(fun, cntxt) {
    if (! suppressUndef(fun, cntxt)) {
        msg <- gettextf("no visible global function definition for '%s'",
                        as.character(fun))
        cntxt$warn(msg, cntxt)
    }
}
```

19b ⟨`notifyUndefVar` *function* 19b⟩≡

```
notifyUndefVar <- function(var, cntxt) {
    if (! suppressUndef(var, cntxt)) {
        msg <- gettextf("no visible binding for global variable '%s'",
                        as.character(var))
        cntxt$warn(msg, cntxt)
    }
}
```

Codetools currently optionally notifies about use of local functions. This is of course not an error but may sometimes be the result of a mis-spelling. For now the compiler does not notify about these, but this could be changed by redefining `notifyLocalFun` .

19c ⟨`notifyLocalFun` *function* 19c⟩≡

```
notifyLocalFun <- function(fun, cntxt) {
    if (! suppressAll(cntxt))
        NULL
}
```

Warnings about possible improper use of `...` and `..n` variables are sent by `notifyWrongDotsUse`.

20a ⟨`notifyWrongDotsUse` *function* 20a⟩≡
```
notifyWrongDotsUse <- function(var, cntxt) {
    if (! suppressAll(cntxt))
        cntxt$warn(paste(var, "may be used in an incorrect context"), cntxt)
}
```

Wrong argument count issues are signaled by `notifyWrongArgCount`.

20b ⟨`notifyWrongArgCount` *function* 20b⟩≡
```
notifyWrongArgCount <- function(fun, cntxt) {
    if (! suppressAll(cntxt))
        cntxt$warn(gettextf("wrong number of arguments to '%s'",
                            as.character(fun)),
                   cntxt)
}
```

Other issues with calls that do not match their definitions are signaled by `notifyBadCall`. Ideally these should be broken down more finely, but that would require some rewriting of the error signaling in `match.call`.

20c ⟨`notifyBadCall` *function* 20c⟩≡
```
notifyBadCall <- function(w, cntxt) {
    if (! suppressAll(cntxt))
        cntxt$warn(w, cntxt)
}
```

`break` or `next` calls that occur in a context where no loop is visible will most likely result in runtime errors, and `notifyWrongBreakNext` is used to signal such cases.

20d ⟨`notifyWrongBreakNext` *function* 20d⟩≡
```
notifyWrongBreakNext <- function(fun, cntxt) {
    if (! suppressAll(cntxt)) {
        msg <- paste(fun, "may be used in wrong context: no loop is visible")
        cntxt$warn(msg, cntxt)
    }
}
```

Several issues can arise in assignments. For super-assignments a target variable should be defined; otherwise there will be a runtime warning.

20e ⟨`notifyNoSuperAssignVar` *function* 20e⟩≡
```
notifyNoSuperAssignVar <- function(symbol, cntxt) {
    if (! suppressAll(cntxt)) {
        msg <- gettextf("no visible binding for '<<-' assignment to '%s'",
                        as.character(symbol))
        cntxt$warn(msg, cntxt)
    }
}
```

If the compiler detects an invalid function in a complex assignment then this is signaled at compile time; a corresponding error would occur at runtime.

20f ⟨`notifyBadAssignFun` *function* 20f⟩≡

```
notifyBadAssignFun <- function(fun, cntxt) {
    if (! suppressAll(cntxt))
        cntxt$warn(gettext("invalid function in complex assignment"))
}
```

In `switch` calls it is an error if a character selector argument is used and there are multiple default alternatives. The compiler signals a possible problem with `notifyMultipleSwitchDefaults` if there are some named cases but more than one unnamed ones.

21 ⟨`notifyMultipleSwitchDefaults` *function* 21⟩≡

```
notifyMultipleSwitchDefaults <- function(ndflt, cntxt)
    if (! suppressAll(cntxt))
        cntxt$warn(gettext("more than one default provided in switch() call"),
                   cntxt)
```

5 Compilation environments

At this point the compiler will essentially use the interpreter to evaluate an expression of the form

```
if (x > 0) log(x) else 0
```

since `if` is a `SPECIAL` function. To make further improvements the compiler needs to be able to implement the `if` expression in terms of conditional and unconditional branch instructions. It might then also be useful to implement `>` and `log` with special virtual machine instructions. To be able to do this, the compiler needs to know that `if`, `>`, and `log` refer to the standard versions of these functions in the base package. While this is very likely, it is not guaranteed.

R is a very dynamic language. Functions defined in the base and other packages could be shadowed at runtime by definitions in loaded user packages or by local definitions within a function. It is even possible for user code to redefine the functions in the base package, though this is discouraged by binding locking and would be poor programming practice. Finally, it is possible for functions called prior to evaluating an expression like the one above to reach into their calling environment and add new definitions of `log` or `if` that wound then be used in evaluating this expression. Again this is not common and generally not a good idea outside of a debugging context.

Ideally the compiler should completely preserve semantics of the language implemented by the interpreter. While this is possible it would significantly complicate the compiler and the compiled code, and carry at least some runtime penalty. The approach taken here is therefore to permit the compiler to inline some functions when they are not visibly shadowed in the compiled code. What the compiler is permitted to do is determined by the setting of an optimization level. The details are desctibed in Section 6.

For the compiler to be able to decide whether is can inline a function it needs to be able to determine whether there are any local variable that might shadow a variable from a base package. This requires adding environment information to the compilation process.

5.1 Representing compilation environments

When compiling an expression the compiler needs to take into account an evaluation environment, which would typically be a toplevel environment, along with local variable definitions discovered during the compilation process. The evaluation environment should not be modified, so the local variables need to be considered in addition to ones defined in the evaluation environment. If an expression

```
{ x <- 1; x + 2 }
```

is compiled for evaluation in the global environment then existing definitions in the global environment as well as the new definition for `x` need to be taken into account. To address this the compilation environment is a list of two components, an environment and a list of character vectors. The environment consists of one frame for each level of local variables followed by the top level evaluation environment. The list of character vectors consist of one element for each frame for which local variables have been discovered by the compiler. For efficiency the compilation environment structure also includes a character vector `ftype` classifying each frame as a local, namespace, or global frame.

22a ⟨`makeCenv` *function* 22a⟩≡

```
## Create a new compiler environment
## **** need to explain the structure
makeCenv <- function(env) {
    structure(list(extra = list(character(0)),
                   env = env,
                   ftypes = frameTypes(env)),
              class = "compiler_environment")
}
```

When an expression is to be compiled in a particular environment a first step is to identify any local variable definitions and add these to the top level frame.

22b ⟨`addCenvVars` *function* 22b⟩≡

```
## Add vars to the top compiler environment frame
addCenvVars <- function(cenv, vars) {
    cenv$extra[[1]] <- union(cenv$extra[[1]], vars)
    cenv
}
```

When compiling a function a new frame is added to the compilation environment. Typically a number of local variables are added immediately, so an optional `vars` argument is provided so this can be done without an additional call to `addCenvVars`.

22c ⟨`addCenvFrame` *function* 22c⟩≡

```
## Add a new frame to a compiler environment
addCenvFrame <- function(cenv, vars) {
    cenv$extra <- c(list(character(0)), cenv$extra)
    cenv$env <- new.env(parent = cenv$env)
    cenv$ftypes <- c("local", cenv$ftypes)
    if (missing(vars))
        cenv
    else
        addCenvVars(cenv, vars)
}
```

The compilation environment is queried by calling `findCenvVar`. If a binding for the specified variable is found then `findCenvVar` returns a list containing information about the binding. If no binding is found then `NULL` is returned.

23a ⟨`findCenvVar` *function* 23a⟩≡

```
## Find binding information for a variable (character or name).
## If a binding is found, return a list containing components
##   ftype -- one of "local", "namespace", "global"
##   value -- current value if available
##   frame -- frame containing the binding (not useful for "local" variables)
##   index -- index of the frame (1 for top, 2, for next one, etc.)
## Return NULL if no binding is found.
## **** drop the index, maybe value, to reduce cost? (query as needed?)
findCenvVar <- function(var, cenv) {
    if (typeof(var) == "symbol")
        var <- as.character(var)
    extra <- cenv$extra
    env <- cenv$env
    frame <- NULL
    ⟨search extra entries and environment frames 23b⟩
    ⟨search the remaining environment frames if necessary 23c⟩
    ⟨create the findCenvVar result 24a⟩
}
```

The initial search for a matching binding proceeds down each frame for which there is also an entry in `extra`, searching the `extra` entry before the environment frame.

23b ⟨*search* `extra` *entries and environment frames* 23b⟩≡

```
for (i in seq_along(cenv$extra)) {
    if (var %in% extra[[i]] || exists(var, env, inherits = FALSE)) {
        frame <- env
        break
    }
    else
        env <- parent.env(env)
}
```

If `frame` is still `NULL` after the initial search then the remaining environment frames from the evaluation environment for which there are no corresponding entries in `extra` are searched.

23c ⟨*search the remaining environment frames if necessary* 23c⟩≡

```
if (is.null(frame)) {
    empty <- emptyenv()
    while (! identical(env, empty)) {
        i <- i + 1
        if (exists(var, env, inherits = FALSE)) {
            frame <- env
            break
        }
        else
            env <- parent.env(env)
```

```
    }
}
```

If a binding frame is found then the result consists of a list containing the frame, the frame type, the value if available, and the frame index. The value is not looked up for `...` variables. A promise to compute the value is stored in an environment in the result. This avoids computing the value in some cases where doing so may fail or produce unwanted side effects.

24a ⟨*create the* `findCenvVar` *result* 24a⟩≡

```
if (! is.null(frame)) {
    if (exists(var, frame, inherits = FALSE) && var != "...") {
        value <- new.env(parent = emptyenv())
        delayedAssign("value", get(var, frame, inherits = FALSE),
                      assign.env = value)
    }
    else
        value <- NULL
    list(frame = frame, ftype = cenv$ftypes[i], value = value, index = i)
}
else
    NULL
```

Useful functions for querying the environment associated with a compilation context are `findVar`, `findLocVar`, and `findFunDef`. The function `findVar` returns `TRUE` is a binding for the specified variable is visible and `FALSE` otherwise.

24b ⟨`findVar` *function* 24b⟩≡

```
findVar <- function(var, cntxt) {
    cenv <- cntxt$env
    info <- findCenvVar(var, cenv)
    ! is.null(info)
}
```

`findLocVar` returns `TRUE` only if a local binding is found.

24c ⟨`findLocVar` *function* 24c⟩≡

```
## test whether a local version of a variable might exist
findLocVar <- function(var, cntxt) {
    cenv <- cntxt$env
    info <- findCenvVar(var, cenv)
    ! is.null(info) && info$ftype == "local"
}
```

`findFunDef` returns a function definition if one is available for the specified name and `NULL` otherwise.

24d ⟨`findFunDef` *function* 24d⟩≡

```
## **** should this check for local functions as well?
findFunDef <- function(fun, cntxt) {
    cenv <- cntxt$env
    info <- findCenvVar(fun, cenv)
    if (! is.null(info$value) && is.function(info$value$value))
        info$value$value
```

```
    else
        NULL
}
```

5.2 Identifying possible local variables

For the compiler to be able to know that it can optimize a reference to a particular global function or variable it needs to be able to determine that that variable will not be shadowed by a local definition at runtime. R semantics do not allow certain identification of local variables. If a function body consist of the two lines

```
if (x) y <- 1
y
```

then whether the variable `y` in the second line is local or global depends on the value of `x`. Lazy evaluation of arguments also means what whether and when an assignment in a function argument occurs can be uncertain.

The approach taken by the compiler is to conservatively identify all variables that might be created within an expression, such as a function body, and consider those to be potentially local variables that inhibit optimizations. This ignores runtime creation of new variables, but as already mentioned that is generally not good programming practice.

Variables are created by the assignment operators `<-` and `=` and by `for` loops. In addition, calls to `assign` and `delayedAssign` with a literal character name argument are considered to create potential local variables if the environment argument is missing, which means the assignment is in the current environment.

A simple approach for identifying all local variables created within an expression is given by

25a ⟨*findlocals0* 25a⟩≡
```
findLocals0 <- function(e) {
    if (typeof(e) == "language") {
        if (typeof(e[[1]]) %in% c("symbol", "character"))
            switch(as.character(e[[1]]),
                   ⟨findLocals0 switch clauses 25b⟩
                   findLocalsList0(e[-1]))
         else findLocalsList0(e)
    }
    else character(0)
}

findLocalsList0 <- function(elist)
    unique(unlist(lapply(elist, findLocals0)))
```

For assignment expressions the assignment variable is added to any variables found in the value expression.

25b ⟨`findLocals0` *switch clauses* 25b⟩≡
```
"=" =,
"<-" = unique(c(getAssignedVar(e),
                findLocalsList0(e[-1]))),
```

The assigned variable is determined by `getAssignedVar`:

26a ⟨`getAssignedVar` *function* 26a⟩≡
```
getAssignedVar <- function(e) {
    v <- e[[2]]
    if (missing(v))
        stop(gettextf("bad assignment: %s", pasteExpr(e)),
             domain = NA)
    else if (typeof(v) %in% c("symbol", "character"))
        as.character(v)
    else {
        while (typeof(v) == "language") {
            if (length(v) < 2)
                stop(gettextf("bad assignment: %s", pasteExpr(e)),
                     domain = NA)
            v <- v[[2]]
            if (missing(v))
                stop(gettextf("bad assignment: %s", pasteExpr(e)),
                     domain = NA)
        }
        if (typeof(v) != "symbol")
            stop(gettextf("bad assignment: %s", pasteExpr(e)),
                 domain = NA)
        as.character(v)
    }
}
```

For `for` loops the loop variable is added to any variables found in the sequence and body expressions.

26b ⟨`findLocals0` *switch clauses* 25b⟩+≡
```
"for" = unique(c(as.character(e[2]),
                 findLocalsList0(e[-2]))),
```

The variable in `assign` and `delayedAssign` expressions is considered local if it is an explicit character string and there is no environment argument.

26c ⟨`findLocals0` *switch clauses* 25b⟩+≡
```
"delayedAssign" =,
"assign" = if (length(e) == 3 &&
               is.character(e[[2]]) &&
               length(e[[2]]) == 1)
               c(e[[2]], findLocals0(e[[3]], shadowed))
           else findLocalsList1(e[1], shadowed),
```

Variables defined within local functions created by `function` expressions do not shadow globals within the containing expression and therefore `function` expressions do not contribute any new local variables. Similarly, `local` calls without an environment argument create a new environment for evaluating their expression and do not add new local variables. If an environment argument is present then this might be the current environment and so assignments in the expression are considered to create possible local variables. Finally, `~`, `expression`, and `quote` do not evaluate their arguments and so do not contribute new local variables.

27a ⟨findLocals0 *switch clauses* 25b⟩+≡
```
"function" = character(0),
"~" = character(0),
"local" = if (length(e) == 2) character(0)
          else findLocalsList0(e[-1]),
"expression" =,
"quote" = character(0),
```

Other functions, for example `Quote` from the `methods` package, are also known to not evaluate their arguments but these do not often contain assignment expressions and so ignoring them only slightly increases the degree of conservatism in this approach.

A problem with this simple implementation is that it assumes that all of the functions named in the `switch` correspond to the bindings in the base package. This is reasonable for the ones that are syntactically special, but not for `expression`, `local` and `quote`. These might be shadowed by local definitions in a surrounding function. To allow for this we can add an optional variable `shadowed` for providing a character vector of names of variables with shadowing local definitions.

27b ⟨findLocals1 *function* 27b⟩≡
```
findLocals1 <- function(e, shadowed = character(0)) {
    if (typeof(e) == "language") {
        if (typeof(e[[1]]) %in% c("symbol", "character")) {
            v <- as.character(e[[1]])
            switch(v,
                   ⟨findLocals1 switch clauses 27d⟩
                   findLocalsList1(e[-1], shadowed))
        }
         else findLocalsList1(e, shadowed)
    }
    else character(0)
}
```

27c ⟨findLocalsList1 *function* 27c⟩≡
```
findLocalsList1 <- function(elist, shadowed)
    unique(unlist(lapply(elist, findLocals1, shadowed)))
```

The handling of assignment operators, `for` loops, `function` and `~` expressions is analogous to the approach in `findLocals0`.

27d ⟨findLocals1 *switch clauses* 27d⟩≡
```
"=" =,
"<-" = unique(c(getAssignedVar(e),
                findLocalsList1(e[-1], shadowed))),
"for" = unique(c(as.character(e[2]),
                 findLocalsList1(e[-2], shadowed))),
"delayedAssign" =,
"assign" = if (length(e) == 3 &&
               is.character(e[[2]]) &&
               length(e[[2]]) == 1)
               c(e[[2]], findLocals1(e[[3]], shadowed))
```

```
            else findLocalsList1(e[1], shadowed),
"function" = character(0),
"~" = character(0),
```

The rules for ignoring assignments in `local`, `expression`, and `quote` calls are only applied if there are no shadowing definitions.

28a ⟨`findLocals1` *switch clauses* 27d⟩+≡
```
"local" = if (! v %in% shadowed && length(e) == 2)
              character(0)
          else findLocalsList1(e[-1], shadowed),
"expression" =,
"quote" = if (! v %in% shadowed)
              character(0)
          else findLocalsList1(e[-1], shadowed),
```

The assignment functions could also be shadowed, but this is not very common, and assuming that they are not errs in the conservative direction.

This approach can handle the case where `quote` or one of the other non-syntactic functions is shadowed by an outer definition but does not handle assignments that occur in the expression itself. For example, in

```
function (f, x, y) {
    local <- f
    local(x <- y)
    x
}
```

the reference to `x` in the third line has to be considered potentially local. To deal with this multiple passes are needed. The first pass assumes that `expression`, `local` or `quote` might be shadowed by local assignments. If no assignments to some of them are visible, then a second pass can be used in which they are assumed not to be shadowed. This can be iterated to convergence. It is also useful to check before returning whether any of the syntactically special variables has been assigned to. If so, so a warning is issued.

28b ⟨`findLocalsList` *function* 28b⟩≡
```
findLocalsList <- function(elist, cntxt) {
    initialShadowedFuns <- c("expression", "local", "quote")
    shadowed <- Filter(function(n) ! isBaseVar(n, cntxt), initialShadowedFuns)
    specialSyntaxFuns <- c("~", "<-", "=", "for", "function")
    sf <- initialShadowedFuns
    nsf <- length(sf)
    repeat {
        vals <- findLocalsList1(elist, sf)
        redefined <- sf %in% vals
        last.nsf <- nsf
        sf <- unique(c(shadowed, sf[redefined]))
        nsf <- length(sf)
        ## **** need to fix the termination condition used in codetools!!!
        if (last.nsf == nsf) {
            rdsf <- vals %in% specialSyntaxFuns
```

```
            if (any(rdsf)) {
                msg <- ngettext(sum(rdsf),
                                "local assignment to syntactic function: ",
                                "local assignments to syntactic functions: ")
                warning(msg, paste(vals[rdsf], collapse = ", "),
                        domain = NA)
            }
            return(vals)
        }
    }
}
```

29a ⟨findLocals *function* 29a⟩≡

```
findLocals <- function(e, cntxt)
    findLocalsList(list(e), cntxt)
```

Standard definitions for all functions in `initialShadowedFuns` are in the base package and `isBaseVar` checks the compilation environment to see whether the specified variable's definition comes from that package either via a namespace or the global environment.

29b ⟨isBaseVar *function* 29b⟩≡

```
isBaseVar <- function(var, cntxt) {
    info <- getInlineInfo(var, cntxt)
    (! is.null(info) &&
     (identical(info$frame, .BaseNamespaceEnv) ||
      identical(info$frame, baseenv())))
}
```

The use of `getInlineInfo`, defined in Section 6, means that the setting of the `optimize` compiler option will influence whether a variable should be considered to be from the base package or not. It might also be useful to warn about assignments to other functions.

When a `function` expression is compiled, its body and default arguments need to be compiled using a compilation environment that contains a new frame for the function that contains variables for the arguments and any assignments in the body and the default expressions. `funEnv` creates such an environment.

29c ⟨funEnv *function* 29c⟩≡

```
## augment compiler environment with function args and locals
funEnv <- function(forms, body, cntxt) {
    cntxt$env <- addCenvFrame(cntxt$env, names(forms))
    locals <- findLocalsList(c(forms, body), cntxt)
    addCenvVars(cntxt$env, locals)
}
```

6 The inlining mechanism

To allow for inline coding of calls to some functions the `cmpCall` function calls the `tryInline` function. The `tryInline` function will either generate code for the call and return `TRUE`, or it will decline to do so and return `FALSE`, in which case the standard code generation process for a function call is used.

The function `tryInline` calls `getInlineInfo` to determine whether inlining is permissible given the current environment and optimization settings. There are four possible optimization levels:

Level 0: No inlining.

Level 1: Functions in the base packages found through a namespace that are not shadowed by function arguments or visible local assignments may be inlined.

Level 2: In addition to the inlining permitted by Level 1, functions that are syntactically special or are considered core language functions and are found via the global environment at compile time may be inlined.

Level 3: Any function in the base packages found via the global environment may be inlined.

The syntactically special and core language functions are

30a ⟨`languageFuns` *definition* 30a⟩≡

```
languageFuns <- c("^", "~", "<", "<<-", "<=", "<-", "=", "==", ">", ">=",
                  "|", "||", "-", ":", "!", "!=", "/", "(", "[", "[<-", "[[",
                  "[[<-", "{", "@", "$", "$<-", "*", "&", "&&", "%/%", "%*%",
                  "%%", "+",
                  "::", ":::", "@<-",
                  "break", "for", "function", "if", "next", "repeat", "while",
                  "local", "return", "switch")
```

The default optimization level is Level 2. Future versions of the compiler may allow some functions to be explicitly excluded from inlining and may provide a means for allowing user-defined functions to be declared eligible for inlining.

If inlining is permissible then the result returned by `getInlineInfo` contains the packages associated with the specified variable in the current environment. The variable name and package are then looked up in a data base of handlers. If a handler is found then the handler is called. The handler can either generate code and return `TRUE` or decline to and return `FALSE`. If inlining is not possible then `getInlineInfo` returns `NULL` and `tryInline` returns `FALSE`.

30b ⟨`tryInline` *function* 30b⟩≡

```
tryInline <- function(e, cb, cntxt) {
    name <- as.character(e[[1]])
    info <- getInlineInfo(name, cntxt)
    if (is.null(info))
        FALSE
    else {
        h <- getInlineHandler(name, info$package)
        if (! is.null(h))
            h(e, cb, cntxt)
        else FALSE
    }
}
```

The function `getInlineInfo` implements the optimization rules described at the beginning of this section.

31a ⟨`getInlineInfo` *function* 31a⟩≡

```
getInlineInfo <- function(name, cntxt) {
    optimize <- cntxt$optimize
    if (optimize > 0) {
        info <- findCenvVar(name, cntxt$env)
        if (is.null(info))
            NULL
        else {
            ftype <- info$ftype
            frame <- info$frame
            if (ftype == "namespace") {
                ⟨fixup for a namespace import frame 31b⟩
                info$package <- nsName(findHomeNS(name, frame))
                info
            }
            else if (ftype == "global" &&
                     (optimize >= 3 ||
                      (optimize >= 2 && name %in% languageFuns))) {
                info$package <- packFrameName(frame)
                info
            }
            else NULL
        }
    }
    else NULL
}
```

The code for finding the home namespace from a namespace import frame is needed here to deal with the fact that a namespace may not be registered when this function is called, so the mechanism used in `findHomeNS` to locate the namespace to which an import frame belongs may not work.

31b ⟨*fixup for a namespace import frame* 31b⟩≡

```
if (! isNamespace(frame)) {
    ## should be the import frame of the current topenv
    top <- topenv(cntxt$env$env)
    if (! isNamespace(top) ||
        ! identical(frame, parent.env(top)))
        cntxt$stop(gettext("bad namespace import frame"))
    frame <- top
}
```

For this version of the compiler the inline handler data base is managed as an environment in which handlers are entered and looked up by name. For now it is assumed that a name can only appear associated with one package and an error is signaled if an attempt is made to redefine a handler for a given name for a different package than an existing definition. This can easily be changed if it should prove too restrictive.

31c ⟨*inline handler implementation* 31c⟩≡

```
inlineHandlers <- new.env(hash = TRUE, parent = emptyenv())

setInlineHandler <- function(name, h, package = "base") {
    if (exists(name, inlineHandlers, inherits = FALSE)) {
        entry <- get(name, inlineHandlers)
        if (entry$package != package) {
            fmt <- "handler for '%s' is already defined for another package"
            stop(gettextf(fmt, name), domain = NA)
        }
    }
    entry <- list(handler = h, package = package)
    assign(name, entry, inlineHandlers)
}

getInlineHandler <- function(name, package = "base") {
    if (exists(name, inlineHandlers, inherits = FALSE)) {
        hinfo <- get(name, inlineHandlers)
        if (hinfo$package == package)
            hinfo$handler
        else NULL
    }
    else NULL
}

haveInlineHandler <- function(name, package = "base") {
    if (exists(name, inlineHandlers, inherits = FALSE)) {
        hinfo <- get(name, inlineHandlers)
        package == hinfo$package
    }
    else FALSE
}
```

7 Default inlining rules for primitives

This section defines generic handlers for `BUILTIN` and `SPECIAL` functions. These are installed programmatically for all `BUILTIN` and `SPECIAL` functions. The following sections present more specialized handlers for a range of functions that are installed in place of the default ones.

32 ⟨*install default inlining handlers* 32⟩≡

```
local({
    ⟨install default SPECIAL handlers 35b⟩
    ⟨install default BUILTIN handlers 34c⟩
})
```

The handler installations are wrapped in a `local` call to reduce environment pollution.

7.1 BUILTIN functions

Calls to functions known at compile time to be of type BUILTIN can be handled more efficiently. The interpreter evaluates all arguments for BUILTIN functions before calling the function, so the compiler can evaluate the arguments in line without the need for creating promises.

A generic handler for inlining a call to a BUILIN function is provided by cmpBuiltin. For now, the handler returns FALSE if the call contains missing arguments, which are currently not allowed in BUILTIN functions, or ... arguments. The handling of ... arguments should be improved. For BUILTIN functions the function to call is pushed on the stack with the GETBUILTIN instruction. The internal argument allows cmpBuiltin to be used with .Internal functions of type BUILTIN as well; this is used in the handler for .Internal defined in Section 8.4.

33a ⟨cmpBuiltin *function* 33a⟩≡
```
cmpBuiltin <- function(e, cb, cntxt, internal = FALSE) {
    fun <- e[[1]]
    args <- e[-1]
    names <- names(args)
    if (dots.or.missing(args))
        FALSE
    else {
        ci <- cb$putconst(fun)
        if (internal)
            cb$putcode(GETINTLBUILTIN.OP, ci)
        else
            cb$putcode(GETBUILTIN.OP, ci)
        cmpBuiltinArgs(args, names, cb, cntxt)
        ci <- cb$putconst(e)
        cb$putcode(CALLBUILTIN.OP, ci)
        if (cntxt$tailcall) cb$putcode(RETURN.OP)
        TRUE
    }
}
```

Argument evaluation code is generated by cmpBuiltinArgs. In the context of BUILTIN functions missing arguments are currently not allowed. But to allow cmpBuiltinArgs to be used in other contexts missing arguments are supported if the optional argument missingOK is TRUE.

33b ⟨cmpBuiltinArgs *function* 33b⟩≡
```
cmpBuiltinArgs <- function(args, names, cb, cntxt, missingOK = FALSE) {
    ncntxt <- make.argContext(cntxt)
    for (i in seq_along(args)) {
        a <- args[[i]]
        n <- names[[i]]
        ⟨compile missing BUILTIN argument 34a⟩
        ## **** handle ... here ??
        ⟨signal an error for promise or bytecode argument 11d⟩
        ⟨compile a general BUILTIN argument 34b⟩
    }
}
```

Missing argument code is generated by

34a ⟨*compile missing* BUILTIN *argument* 34a⟩≡
```
if (missing(a)) {
    if (missingOK) {
        cb$putcode(DOMISSING.OP)
        cmpTag(n, cb)
    }
    else
        cntxt$stop(gettext("missing arguments are not allowed"), cntxt)
}
```

The error case should not be reached as `cmpBuiltinArgs` should not be called with missing arguments unless `missingOK` is `TRUE`.

The code for general arguments handles symbols separately to allow for the case when missing values are acceptable. Constant folding is tried first since the constant folding code in `cmp` is not reached in this case. Constant folding is needed here since it doesn't go through `cmp`.

34b ⟨*compile a general* BUILTIN *argument* 34b⟩≡
```
else {
    if (is.symbol(a)) {
        ca <- constantFold(a, cntxt)
        if (is.null(ca)) {
            cmpSym(a, cb, ncntxt, missingOK)
            cb$putcode(PUSHARG.OP)
        }
        else
            cmpConstArg(ca$value, cb, cntxt)
    }
    else if (typeof(a) == "language") {
        cmp(a, cb, ncntxt)
        cb$putcode(PUSHARG.OP)
    }
    else
        cmpConstArg(a, cb, cntxt)
    cmpTag(n, cb)
}
```

Handling the constant case separately is not really necessary but makes the code a bit cleaner.

Default handlers for all `BUILTIN` functions in the `base` package are installed programmatically by

34c ⟨*install default* BUILTIN *handlers* 34c⟩≡
```
for (b in basevars[types == "builtin"])
    if (! haveInlineHandler(b, "base"))
        setInlineHandler(b, cmpBuiltin)
```

7.2 SPECIAL functions

Calls to functions known to be of type `SPECIAL` can also be compiled somewhat more efficiently by the `cmpSpecial` function:

35a ⟨cmpSpecial *function* 35a⟩≡
```
cmpSpecial <- function(e, cb, cntxt) {
    fun <- e[[1]]
    if (typeof(fun) == "character")
        fun <- as.name(fun)
    ci <- cb$putconst(e)
    cb$putcode(CALLSPECIAL.OP, ci)
    if (cntxt$tailcall)
        cb$putcode(RETURN.OP)
    TRUE
}
```

This handler is installed for all `SPECIAL` functions in the base package with

35b ⟨*install default* `SPECIAL` *handlers* 35b⟩≡
```
basevars <- ls('package:base', all = TRUE)
types <- sapply(basevars, function(n) typeof(get(n)))
for (s in basevars[types == "special"])
    if (! haveInlineHandler(s, "base"))
        setInlineHandler(s, cmpSpecial)
```

8 Some simple inlining handlers

This section presents inlining handlers for a number of core primitive functions. With these additions the compiler will begin to show some performance improvements.

8.1 The left brace sequencing function

The inlining handler for `{` needs to consider that a pair of braces `{` and `}` can surround zero, one, or more expressions. A set of empty braces is equivalent to the constant `NULL`. If there is more than one expression, then all the values of all expressions other than the last are ignored. These expressions are compiled in a no-value context (currently equivalent to a non-tail-call context), and then code is generated to pop their values off the stack. The final expression is then compiled according to the context in which the braces expression occurs.

35c ⟨*inlining handler for left brace function* 35c⟩≡
```
setInlineHandler("{", function(e, cb, cntxt) {
    n <- length(e)
    if (n == 1)
        cmp(NULL, cb, cntxt)
    else {
        if (n > 2) {
            ncntxt <- make.noValueContext(cntxt)
            for (i in 2 : (n - 1)) {
                cmp(e[[i]], cb, ncntxt)
                cb$putcode(POP.OP)
            }
        }
```

```
        cmp(e[[n]], cb, cntxt)
    }
    TRUE
})
```

8.2 The closure constructor function

Compiling of `function` expressions is somewhat similar to compiling promises for function arguments. The body of a function is compiled into a separate byte code object and stored in the constant pool together with the formals. Then code is emitted for creating a closure from the formals, compiled body, and the current environment. For now, only the body of functions is compiled, not the default argument expressions. This should be changed in future versions of the compiler.

36 ⟨*inlining handler for* `function` 36⟩≡

```
setInlineHandler("function", function(e, cb, cntxt) {
    forms <- e[[2]]
    body <- e[[3]]
    ncntxt <- make.functionContext(cntxt, forms, body)
    cbody <- genCode(body, ncntxt)
    ci <- cb$putconst(list(forms, cbody))
    cb$putcode(MAKECLOSURE.OP, ci)
    if (cntxt$tailcall) cb$putcode(RETURN.OP)
    TRUE
})
```

8.3 The left parenthesis function

In R an expression of the form `(expr)` is interpreted as a call to the function `(` with the argument `expr`. Parentheses are used to guide the parser, and for the most part `(expr)` is equivalent to `expr`. There are two exceptions:

- Since `(` is a function an expression of the form `(...)` is legal whereas just `...` may not be, depending on the context. A runtime error will occur unless the `...` argument expands to exactly one non-missing argument.
- In tail position a call to `(` sets the visible flag to `TRUE`. So at top level for example the result of an assignment expression `x <- 1` would not be printed, but the result of `(x <- 1` would be printed. It is not clear that this feature really needs to be preserved within functions — it could be made a feature of the read-eval-print loop — but for now it is a feature of the interpreter that the compiler should preserve.

The inlining handler for `(` calls handles a `...` argument case or a case with fewer or more than one argument as a generic `BUILTIN` call. If the expression is in tail position then the argument is compiled in a non-tail-call context, a `VISIBLE` instruction is emitted to set the visible flag to `TRUE`, and a RETURN instruction is emitted. If the expression is in non-tail position, then code for the argument is generated in the current context.

37a ⟨*inlining handler for* (37a⟩≡
```
setInlineHandler("(", function(e, cb, cntxt) {
    if (any.dots(e))
        cmpBuiltin(e, cb, cntxt) ## punt
    else if (length(e) != 2) {
        notifyWrongArgCount("(", cntxt)
        cmpBuiltin(e, cb, cntxt) ## punt
    }
    else if (cntxt$tailcall) {
        ncntxt <- make.nonTailCallContext(cntxt)
        cmp(e[[2]], cb, ncntxt)
        cb$putcode(VISIBLE.OP)
        cb$putcode(RETURN.OP)
        TRUE
    }
    else {
        cmp(e[[2]], cb, cntxt)
        TRUE
    }
})
```

8.4 The .Internal function

One frequently used SPECIAL function is .Internal. When the .Internal function called is of type BUILTIN it is useful to compile the call as for a BUILTIN function. For .Internal functions of type SPECIAL there is less of an advantage, and so the .Internal expression is compiled with cmpSpecial. It may be useful to use introduce a GETINTLSPECIAL instruction and handle these analogously to .Internal functions of type BUILTIN.

37b ⟨*inlining handler for* .Internal 37b⟩≡
```
setInlineHandler(".Internal", function(e, cb, cntxt) {
    ee <- e[[2]]
    sym <- ee[[1]]
    if (.Internal(is.builtin.internal(sym)))
        cmpBuiltin(ee, cb, cntxt, internal = TRUE)
    else
        cmpSpecial(e, cb, cntxt)
})
```

8.5 The local function

While local is currently implemented as a closure, because of its importance relative to local variable determination it is a good idea to inline it as well. The current semantics are such that the interpreter treats

```
local(expr)
```

essentially the same as

```
(function() expr)()
```

There may be some minor differences related to what the `sys.xyz` functions return but these do not seem to be important. So the compiler handles one argument `local` calls by making this conversion and compiling the result.

38a ⟨*inlining handler for* `local` *function* 38a⟩≡

```
setInlineHandler("local", function(e, cb, cntxt) {
    if (length(e) == 2) {
        ee <- as.call(list(as.call(list(as.name("function"), NULL, e[[2]]))))
        cmp(ee, cb, cntxt)
        TRUE
    }
    else FALSE
})
```

8.6 The `return` function

A call to `return` causes a return from the associated function call, as determined by the lexical context in which the `return` expression is defined. If the `return` is captured in a closure and is executed within a callee then this requires a `longjmp`. A `longjmp` is also needed if the `return` call occurs within a loop that is compiled to a separate code object to support a `setjmp` for `break` or `next` calls. The `RETURNJMP` instruction is provided for this purpose. In all other cases an ordinary `RETURN` instruction can be used. `return` calls with `...`, which may be legal if `...` contains only one argument, or missing arguments or more than one argument, which will produce runtime errors, are compiled as generic `SPECIAL` calls.

38b ⟨*inlining handler for* `return` *function* 38b⟩≡

```
setInlineHandler("return", function(e, cb, cntxt) {
    if (dots.or.missing(e) || length(e) > 2)
        cmpSpecial(e, cb, cntxt) ## **** punt for now
    else {
        if (length(e) == 1)
            val <- NULL
        else
            val <- e[[2]]
        ncntxt <- make.nonTailCallContext(cntxt)
        cmp(val, cb, ncntxt)
        if (cntxt$needRETURNJMP)
            cb$putcode(RETURNJMP.OP)
        else
            cb$putcode(RETURN.OP)
    }
    TRUE
})
```

9 Branching and labels

The code generated so far is straight line code without conditional or unconditional branches. To implement conditional evaluation constructs and loops we need to add conditional and unconditional branching instructions. These make use of the labels mechanism provided by the code buffer.

9.1 Inlining handler for `if` expressions

Using the labels mechanism we can implement an inlining handler for `if` expressions. The first step extracts the components of the expression. An `if` expression with no `else` clause will invisibly return `NULL` if the test is `FALSE`, but the visible flag setting only matters if the `if` expression is in tail position. So the case of no `else` clause will be handled slightly differently in tail and non-tail contexts.

39a ⟨`if` *inline handler body* 39a⟩≡

```
test <- e[[2]]
then.expr <- e[[3]]
if (length(e) == 4) {
    have.else.expr <- TRUE
    else.expr <- e[[4]]
}
else have.else.expr <- FALSE
```

To deal with use of `if (FALSE) ...` for commenting out code and of `if (is.R()) ... else ...` for handling both R and Splus code it is useful to attempt to constant-fold the test. If this succeeds and produces either `TRUE` or `FALSE` then only the appropriate branch is compiled and the handler returns `TRUE`.

39b ⟨`if` *inline handler body* 39a⟩+≡

```
ct <- constantFold(test, cntxt)
if (! is.null(ct) && is.logical(ct$value) && length(ct$value) == 1
    && ! is.na(ct$value)) {
    if (ct$value)
        cmp(then.expr, cb, cntxt)
    else if (have.else.expr)
        cmp(else.expr, cb, cntxt)
    else if (cntxt$tailcall) {
        cb$putcode(LDNULL.OP)
        cb$putcode(INVISIBLE.OP)
        cb$putcode(RETURN.OP)
    }
    else cb$putcode(LDNULL.OP)
    return(TRUE)
}
```

Next, the test code is compiled, a label for the start of code for the `else` clause is generated, and a conditional branch instruction that branches to the `else` label if the test fails is emitted. This is followed by code for the consequent (test is `TRUE`) expression. The `BRIFNOT` takes two operand, the constant pool index for the call and the label to branch to if the value on the stack is `FALSE`. The call is used if an error needs to be signaled for an improper test result on the stack.

40a ⟨`if` *inline handler body* 39a⟩+≡
```
ncntxt <- make.nonTailCallContext(cntxt)
cmp(test, cb, ncntxt)
callidx <- cb$putconst(e)
else.label <- cb$makelabel()
cb$putcode(BRIFNOT.OP, callidx, else.label)
cmp(then.expr, cb, cntxt)
```

The code for the alternative `else` expression will be placed after the code for the consequent expression. If the `if` expression appears in tail position then the code for the consequent will end with a `RETURN` instruction and there is no need to jump over the following instructions for the `else` expression. All that is needed is to record the value of the label for the `else` clause and to emit the code for the `else` clause. If no `else` clause was provided then that code arranges for the value `NULL` to be returned invisibly.

40b ⟨`if` *inline handler body* 39a⟩+≡
```
if (cntxt$tailcall) {
    cb$putlabel(else.label)
    if (have.else.expr)
        cmp(else.expr, cb, cntxt)
    else {
        cb$putcode(LDNULL.OP)
        cb$putcode(INVISIBLE.OP)
        cb$putcode(RETURN.OP)
    }
}
```

On the other hand, if the `if` expression is not in tail position then a label for the next instruction after the `else` expression code is needed, and the consequent expression code needs to end with a `GOTO` instruction to that label. If the expression does not include an `else` clause then the alternative code just places `NULL` on the stack.

40c ⟨`if` *inline handler body* 39a⟩+≡
```
else {
    end.label <- cb$makelabel()
    cb$putcode(GOTO.OP, end.label)
    cb$putlabel(else.label)
    if (have.else.expr)
        cmp(else.expr, cb, cntxt)
    else
        cb$putcode(LDNULL.OP)
    cb$putlabel(end.label)
}
```

The resulting handler definition is

40d ⟨*inlining handler for* `if` 40d⟩≡
```
setInlineHandler("if", function(e, cb, cntxt) {
    ## **** test for missing, ...
```

```
    ⟨if inline handler body 39a⟩
    TRUE
})
```

9.2 Inlining handlers for && and || expressions

In many languages it is possible to convert the expression `a && b` to an equivalent `if` expression of the form

```
if (a) { if (b) TRUE else FALSE }
```

Similarly, in these languages the expression `a || b` is equivalent to

```
if (a) TRUE else if (b) TRUE else FALSE
```

Compilation of these expressions is thus reduced to compiling `if` expressions.

Unfortunately, because of the possibility of `NA` values, these equivalencies do not hold in R. In R, `NA || TRUE` should evaluate to `TRUE` and `NA && FALSE` to `FALSE`. This is handled by introducing special instructions `AND1ST` and `AND2ND` for `&&` expressions and `OR1ST` and `OR2ND` for `||`.

The code generator for `&&` expressions generates code to evaluate the first argument and then emits an `AND1ST` instruction. The `AND1ST` instruction has one operand, the label for the instruction following code for the second argument. If the value on the stack produced by the first argument is `FALSE` then `AND1ST` jumps to the label and skips evaluation of the second argument; the value of the expression is `FALSE`. The code for the second argument is generated next, followed by an `AND2ND` instruction. This removes the values of the two arguments to `&&` from the stack and pushes the value of the expression onto the stack. A `RETURN` instruction is generated if the `&&` expression was in tail position.

41a ⟨*inlining handler for* `&&` 41a⟩≡

```
setInlineHandler("&&", function(e, cb, cntxt) {
    ## **** arity check??
    ncntxt <- make.argContext(cntxt)
    callidx <- cb$putconst(e)
    label <- cb$makelabel()
    cmp(e[[2]], cb, ncntxt)
    cb$putcode(AND1ST.OP, callidx, label)
    cmp(e[[3]], cb, ncntxt)
    cb$putcode(AND2ND.OP, callidx)
    cb$putlabel(label)
    if (cntxt$tailcall)
        cb$putcode(RETURN.OP)
    TRUE
})
```

The code generator for `||` expressions is analogous.

41b ⟨*inlining handler for* `||` 41b⟩≡

```
setInlineHandler("||", function(e, cb, cntxt) {
    ## **** arity check??
    ncntxt <- make.argContext(cntxt)
```

```
    callidx <- cb$putconst(e)
    label <- cb$makelabel()
    cmp(e[[2]], cb, ncntxt)
    cb$putcode(OR1ST.OP, callidx, label)
    cmp(e[[3]], cb, ncntxt)
    cb$putcode(OR2ND.OP, callidx)
    cb$putlabel(label)
    if (cntxt$tailcall)
        cb$putcode(RETURN.OP)
    TRUE
})
```

10 Loops

In principle code for `repeat` and `while` loops can be generated using just `GOTO` and `BRIFNOT` instructions. `for` loops require a little more to manage the loop variable and termination. A complication arises due to the need to support `break` and `next` calls in the context of lazy evaluation of arguments: if a `break` or `next` expression appears in a function argument that is compiled as a closure, then the expression may be evaluated deep inside a series of nested function calls and require a non-local jump. A similar issue arises for calls to the `return` function as described in Section 8.6.

To support these non-local jumps the interpreter sets up a `setjmp` context for each loop, and `break` and `next` use `longjmp` to transfer control. In general, compiled loops need to use a similar approach. For now, this is implemented by the `STARTLOOPCNTXT` and `ENDLOOPCNTXT` instructions. The `STARTLOOPCNTXT` instructions takes one argument, the constant pool index for a code object implementing the loop body. The loop body should end with a call to `ENDLOOPCNTXT`.

At least with some assumptions it is often possible to implement `break` and `next` calls as simple `GOTO`s. If all `break` and `next` calls in a loop can be implemented using `GOTO`s then the loop context is not necessary. The mechanism to enable the simpler code generation is presented in Section 10.4.

The current engine implementation executes one `setjmp` per `STARTLOOPCNTXT` and uses nested calls to `bceval` to run the code. Eventually we should be able to reduce the need for nested `bceval` calls and to arrange that `setjmp` buffers be reused for multiple purposes.

10.1 `repeat` loops

The simplest loop in R is the `repeat` loop. The code generator is defined as

42 ⟨*inlining handler for* `repeat` *loops* 42⟩≡

```
setInlineHandler("repeat", function(e, cb, cntxt) {
    body <- e[[2]]
    ⟨generate context and body code for repeat loop 43a⟩
    ⟨generate repeat and while loop wrap-up code 43c⟩
    TRUE
})
```

If a loop context is not needed then the code for the loop body is just written to the original code buffer. The `else` clause in the code chunk below generates the code for the general case. `genCode` is used to produce a new code object containing code for the loop body followed by the `ENDLOOPCNTXT` instruction, The code object is then installed in the context pool. The `STARTLOOPCNTXT` instruction is emitted to the main code buffer. When a separate code object is written the need for using `RETURNJMP` for `return` calls is indicated by setting the `needRETURNJMP` flag in the compiler context to `TRUE`.

43a ⟨*generate context and body code for* `repeat` *loop* 43a⟩≡

```
if (checkSkipLoopCntxt(body, cntxt))
    cmpRepeatBody(body, cb, cntxt)
else {
    cntxt$needRETURNJMP <- TRUE ## **** do this a better way
    code <- genCode(body, cntxt,
                    function(cb, cntxt) {
                        cmpRepeatBody(body, cb, cntxt)
                        cb$putcode(ENDLOOPCNTXT.OP)
                    })
    bi <- cb$putconst(code)
    cb$putcode(STARTLOOPCNTXT.OP, bi)
}
```

The loop body uses two labels. `loop.label` marks the top of the loop and is the target of the `GOTO` instruction at the end of the body. This label is also used by `next` expressions that do not require `longjmp`s. The `end.loop` label is placed after the `GOTO` instruction and is used by `break` expressions that do not require `longjmp`s. The body is compiled in a context that makes these labels available, and the value left on the stack is removed by a `POP` instruction. The `POP` instruction is followed by a `GOTO` instruction that returns to the top of the loop.

43b ⟨`cmpRepeatBody` *function* 43b⟩≡

```
cmpRepeatBody <- function(body, cb, cntxt) {
    loop.label <- cb$makelabel()
    end.label <- cb$makelabel()
    cb$putlabel(loop.label)
    lcntxt <- make.loopContext(cntxt, loop.label, end.label)
    cmp(body, cb, lcntxt)
    cb$putcode(POP.OP)
    cb$putcode(GOTO.OP, loop.label)
    cb$putlabel(end.label)
}
```

The wrap-up code for the loop places the `NULL` value of the loop expression on the stack and emits `INVISIBLE` and `RETURN` instructions to return the value if the loop appears in tail position.

43c ⟨*generate* `repeat` *and* `while` *loop wrap-up code* 43c⟩≡

```
cb$putcode(LDNULL.OP)
if (cntxt$tailcall) {
    cb$putcode(INVISIBLE.OP)
    cb$putcode(RETURN.OP)
}
```

The `break` and `next` code generators emit `GOTO` instructions if the loop information is available and the `gotoOK` compiler context flag is `TRUE`. A warning is issued if no loop is visible in the compilation context.

44a ⟨*inlining handlers for* `next` *and* `break` 44a⟩≡
```
setInlineHandler("break", function(e, cb, cntxt) {
    if (is.null(cntxt$loop)) {
        notifyWrongBreakNext("break", cntxt)
        cmpSpecial(e, cb, cntxt)
    }
    else if (cntxt$loop$gotoOK) {
        cb$putcode(GOTO.OP, cntxt$loop$end)
        TRUE
    }
    else cmpSpecial(e, cb, cntxt)
})

setInlineHandler("next", function(e, cb, cntxt) {
    if (is.null(cntxt$loop)) {
        notifyWrongBreakNext("next", cntxt)
        cmpSpecial(e, cb, cntxt)
    }
    else if (cntxt$loop$gotoOK) {
        cb$putcode(GOTO.OP, cntxt$loop$loop)
        TRUE
    }
    else cmpSpecial(e, cb, cntxt)
})
```

10.2 `while` loops

The structure for the `while` loop code generator is similar to the structure of the `repeat` code generator:

44b ⟨*inlining handler for* `while` *loops* 44b⟩≡
```
setInlineHandler("while", function(e, cb, cntxt) {
    cond <- e[[2]]
    body <- e[[3]]
    ⟨generate context and body code for while loop 44c⟩
    ⟨generate repeat and while loop wrap-up code 43c⟩
    TRUE
})
```

The context and body generation chunk is similar as well. The expression stored in the code object isn't quite right as what is compiled includes both the test and the body, but this code object should not be externally visible.

44c ⟨*generate context and body code for* `while` *loop* 44c⟩≡
```
if (checkSkipLoopCntxt(cond, cntxt) && checkSkipLoopCntxt(body, cntxt))
    cmpWhileBody(e, cond, body, cb, cntxt)
```

```
else {
    cntxt$needRETURNJMP <- TRUE ## **** do this a better way
    code <- genCode(body, cntxt, ## **** expr isn't quite right
                    function(cb, cntxt) {
                        cmpWhileBody(e, cond, body, cb, cntxt)
                        cb$putcode(ENDLOOPCNTXT.OP)
                    })
    bi <- cb$putconst(code)
    cb$putcode(STARTLOOPCNTXT.OP, bi)
}
```

Again two labels are used, one at the top of the loop and one at the end. The `loop.label` is followed by code for the test. Next is a `BRIFNOT` instruction that jumps to the end of the loop if the value left on the stack by the test is `FALSE`. This is followed by the code for the body, a `POP` instruction, and a `GOTO` instruction that jumps to the top of the loop. Finally, the `end.label` is recorded.

45 ⟨`cmpWhileBody` *function* 45⟩≡

```
cmpWhileBody <- function(call, cond, body, cb, cntxt) {
    loop.label <- cb$makelabel()
    end.label <- cb$makelabel()
    cb$putlabel(loop.label)
    lcntxt <- make.loopContext(cntxt, loop.label, end.label)
    cmp(cond, cb, lcntxt)
    callidx <- cb$putconst(call)
    cb$putcode(BRIFNOT.OP, callidx, end.label)
    cmp(body, cb, lcntxt)
    cb$putcode(POP.OP)
    cb$putcode(GOTO.OP, loop.label)
    cb$putlabel(end.label)
}
```

cmpWhileBody

10.3 `for` loops

Code generation for `for` loops is a little more complex because of the need to manage the loop variable value and stepping through the sequence. Code for `for` loops uses three additional instructions. `STARTFOR` takes the constant pool index of the call, the constant pool index of the loop variable symbol, and the label of the start instruction as operands. It finds the sequence to iterate over on the stack and places information for accessing the loop variable binding and stepping the sequence on the stack before jumping to the label. The call is used if an error for an improper for loop sequence needs to be signaled. The `STEPFOR` instruction takes a label for the top of the loop as its operand. If there are more elements in the sequence then `STEPFOR` advances the position within the sequence, sets the loop variable, and jumps to the top of the loop. Otherwise it drops through to the next instruction. Finally `ENDFOR` cleans up the loop information stored on the stack by `STARTFOR` and leaves the `NULL` loop value on the stack.

The inlining handler for a `for` loop starts out by checking the loop variable and issuing a warning if it is not a symbol. The code generator then declines to inline the loop expression. This means it is compiled as a generic function call and will signal an error at runtime. An alternative would be do generate code to signal the error as is done with improper use of `...` arguments. After checking the symbol, code to compute the sequence to iterate over is generated. From then on the structure is similar to the structure of the other loop code generators.

46a ⟨*inlining handler for* `for` *loops* 46a⟩≡
```
setInlineHandler("for", function(e, cb, cntxt) {
    sym <- e[[2]]
    seq <- e[[3]]
    body <- e[[4]]
    if (! is.name(sym)) {
        ## not worth warning here since the parser should not allow this
        return(FALSE)
    }
    ncntxt <- make.nonTailCallContext(cntxt)
    cmp(seq, cb, ncntxt)
    ci <- cb$putconst(sym)
    callidx <- cb$putconst(e)
    ⟨generate context and body code for for loop 46b⟩
    ⟨generate for loop wrap-up code 47b⟩
    TRUE
})
```

When a `setjmp` context is needed, the label given to `STARTFOR` is just the following instruction, which is a `STARTLOOPCNTXT` instruction. If the context is not needed then the label for the `STARTFOR` instruction will be the loop's `STEPFOR` instruction; if the context is needed then the first instruction in the code object for the body will be a `GOTO` instruction that jumps to the `STEPFOR` instruction. This design means the stepping and the jump can be handled by one instruction instead of two, a step instruction and a `GOTO`.

46b ⟨*generate context and body code for* `for` *loop* 46b⟩≡
```
if (checkSkipLoopCntxt(body, cntxt))
    cmpForBody(callidx, body, ci, cb, cntxt)
else {
    cntxt$needRETURNJMP <- TRUE ## **** do this a better way
    ctxt.label <- cb$makelabel()
    cb$putcode(STARTFOR.OP, callidx, ci, ctxt.label)
    cb$putlabel(ctxt.label)
    code <- genCode(body, cntxt, ## **** expr isn't quite right
                    function(cb, cntxt) {
                        cmpForBody(NULL, body, NULL, cb, cntxt)
                        cb$putcode(ENDLOOPCNTXT.OP)
                    })
    bi <- cb$putconst(code)
    cb$putcode(STARTLOOPCNTXT.OP, bi)
}
```

The body code generator takes an additional argument, the index of the loop label. For the case where a `setjmp` context is needed this argument is `NULL`, and the first instruction generated is a GOTO targeting the STEPFOR instruction. This is labeled by the loop.label label, since this will also be the target used by a next expression. An additional label, body.label is needed for the top of the loop, which is used by STEPFOR if there are more loop elements to process. When the ci argument is not NULL code is being generated for the case without a setjmp context, and the first instruction is the STARTFOR instruction which initializes the loop and jumps to loop.label at the STEPLOOP instruction.

47a ⟨cmpForBody *function* 47a⟩≡
```
cmpForBody <- function(callidx, body, ci, cb, cntxt) {
    body.label <- cb$makelabel()
    loop.label <- cb$makelabel()
    end.label <- cb$makelabel()
    if (is.null(ci))
        cb$putcode(GOTO.OP, loop.label)
    else
        cb$putcode(STARTFOR.OP, callidx, ci, loop.label)
    cb$putlabel(body.label)
    lcntxt <- make.loopContext(cntxt, loop.label, end.label)
    cmp(body, cb, lcntxt)
    cb$putcode(POP.OP)
    cb$putlabel(loop.label)
    cb$putcode(STEPFOR.OP, body.label)
    cb$putlabel(end.label)
}
```

The wrap-up code issues an ENDFOR instruction instead of the LDNULL instruction used for repeat and while loops.

47b ⟨*generate* for *loop wrap-up code* 47b⟩≡
```
cb$putcode(ENDFOR.OP)
if (cntxt$tailcall) {
    cb$putcode(INVISIBLE.OP)
    cb$putcode(RETURN.OP)
}
```

10.4 Avoiding runtime loop contexts

When all uses of break or next in a loop occur only in top level contexts then all break and next calls can be implemented with simple GOTO instructions and a setjmp context for the loop is not needed. Top level contexts are the loop body itself and argument expressions in top level calls to if, {, and (. The switch functions will eventually be included as well. The function checkSkipLoopContxt recursively traverses an expression tree to determine whether all relevant uses of break or next are safe to compile as GOTO instructions. The search returns FALSE if a break or next call occurs in an unsafe place. The search stops and returns TRUE for any expression that cannot contain relevant break or next calls. These stop expressions are calls to the three loop functions and to function. Calls to functions like quote that are known not to evaluate their arguments could also be included among the stop functions but this doesn't seem particularly worth while at this time.

The recursive checking function is defined as

48a ⟨checkSkipLoopCntxt *function* 48a⟩≡
```
checkSkipLoopCntxt <- function(e, cntxt, breakOK = TRUE) {
    if (typeof(e) == "language") {
        fun <- e[[1]]
        if (typeof(fun) == "symbol") {
            fname <- as.character(fun)
            if (! breakOK && fname %in% c("break", "next"))
                FALSE
            else if (isLoopStopFun(fname, cntxt))
                TRUE
            else if (isLoopTopFun(fname, cntxt))
                checkSkipLoopCntxtList(e[-1], cntxt, breakOK)
            else
                checkSkipLoopCntxtList(e[-1], cntxt, FALSE)
        }
        else
            checkSkipLoopCntxtList(e, cntxt, FALSE)
    }
    else TRUE
}
```

A version that operates on a list of expressions is given by

48b ⟨checkSkipLoopCntxtList *function* 48b⟩≡
```
checkSkipLoopCntxtList <- function(elist, cntxt, breakOK) {
    for (a in as.list(elist))
        if (! missing(a) && ! checkSkipLoopCntxt(a, cntxt, breakOK))
            return(FALSE)
    TRUE
}
```

The stop functions are identified by `isLoopStopFun`. This uses `isBaseVar` to ensure that interpreting a reference to a stop function name as referring to the corresponding function in the `base` package is permitted by the current optimization settings.

48c ⟨isLoopStopFun *function* 48c⟩≡
```
isLoopStopFun <- function(fname, cntxt)
    (fname %in% c("function", "for", "while", "repeat") &&
     isBaseVar(fname, cntxt))
```

The top level functions are identified by `isLoopTopFun`. Again the compilation context is consulted to ensure that candidate can be assumed to be from the `base` package.

48d ⟨isLoopTopFun *function* 48d⟩≡
```
isLoopTopFun <- function(fname, cntxt)
    (fname %in% c("(", "{", "if") &&
     isBaseVar(fname, cntxt))
```

The `checkSkipLoopCntxt` function does not check whether calls to `break` or `next` are indeed calls to the `base` functions. Given the special syntactic nature of `break` and `next` this is very unlikely to cause problems, but if it does it will result in some safe loops being considered unsafe and so errs in the conservative direction.

11 More inlining

11.1 Basic arithmetic expressions

The addition and subtraction functions `+` and `-` are `BUILTIN` functions that can both be called with one or two arguments. Multiplication and division functions `*` and `/` require two arguments. Since code generation for all one arguments cases and all two argument cases is very similar these are abstracted out into functions `cmpPrim1` and `cmpPrim2`.

The code generators for addition and subtraction are given by

49a ⟨*inline handlers for* `+` *and* `-` 49a⟩≡
```
setInlineHandler("+", function(e, cb, cntxt) {
    if (length(e) == 3)
        cmpPrim2(e, cb, ADD.OP, cntxt)
    else
        cmpPrim1(e, cb, UPLUS.OP, cntxt)
})

setInlineHandler("-", function(e, cb, cntxt) {
    if (length(e) == 3)
        cmpPrim2(e, cb, SUB.OP, cntxt)
    else
        cmpPrim1(e, cb, UMINUS.OP, cntxt)
})
```

The code generators for multiplication and division are

49b ⟨*inline handlers for* `*` *and* `/` 49b⟩≡
```
setInlineHandler("*", function(e, cb, cntxt)
    cmpPrim2(e, cb, MUL.OP, cntxt))

setInlineHandler("/", function(e, cb, cntxt)
    cmpPrim2(e, cb, DIV.OP, cntxt))
```

Code for instructions corresponding to calls to a `BUILTIN` function with one argument are generated by `cmpPrim1`. The generator produces code for a generic `BUILTIN` call using `cmpBuiltin` if if there are any missing or `...` arguments or if the number of arguments is not equal to one. Otherwise code for the argument is generated in a non-tail-call context, and the instruction provided as the `op` argument is emitted followed by a `RETURN` instruction for an expression in tail position. The `op` instructions take the call as operand for use in error message and for internal dispatching.

49c ⟨`cmpPrim1` *function* 49c⟩≡
```
cmpPrim1 <- function(e, cb, op, cntxt) {
    if (dots.or.missing(e[-1]))
        cmpBuiltin(e, cb, cntxt)
    else if (length(e) != 2) {
        notifyWrongArgCount(e[[1]], cntxt)
        cmpBuiltin(e, cb, cntxt)
    }
    else {
        ncntxt <- make.nonTailCallContext(cntxt)
```

```
        cmp(e[[2]], cb, ncntxt);
        ci <- cb$putconst(e)
        cb$putcode(op, ci)
        if (cntxt$tailcall)
            cb$putcode(RETURN.OP)
        TRUE
    }
}
```

Code generation for the two argument case is similar, except that the second argument has to be compiled with an argument context since the stack already has the value of the first argument on it and that would need to be popped before a jump.

50a ⟨cmpPrim2 *function* 50a⟩≡

```
cmpPrim2 <- function(e, cb, op, cntxt) {
    if (dots.or.missing(e[-1]))
        cmpBuiltin(e, cb, cntxt)
    else if (length(e) != 3) {
        notifyWrongArgCount(e[[1]], cntxt)
        cmpBuiltin(e, cb, cntxt)
    }
    else {
        ncntxt <- make.nonTailCallContext(cntxt)
        cmp(e[[2]], cb, ncntxt);
        ncntxt <- make.argContext(cntxt)
        cmp(e[[3]], cb, ncntxt)
        ci <- cb$putconst(e)
        cb$putcode(op, ci)
        if (cntxt$tailcall)
            cb$putcode(RETURN.OP)
        TRUE
    }
}
```

Calls to the power function `^` and the functions `exp` and `sqrt` can be compiled using `cmpPrim1` and `cmpPrim2` as well:

50b ⟨*inline handlers for* `^`, `exp`, *and* `sqrt` 50b⟩≡

```
setInlineHandler("^", function(e, cb, cntxt)
    cmpPrim2(e, cb, EXPT.OP, cntxt))

setInlineHandler("exp", function(e, cb, cntxt)
    cmpPrim1(e, cb, EXP.OP, cntxt))

setInlineHandler("sqrt", function(e, cb, cntxt)
    cmpPrim1(e, cb, SQRT.OP, cntxt))
```

The `log` function is currently defined as a `SPECIAL`. The inline handler for `log` is thus defined by

50c ⟨*inline handler for* `log` 50c⟩≡

```
setInlineHandler("log", cmpSpecial)
```

It would be a good idea to add instructions for one and two argument log functions.

11.2 Logical operators

Two argument instructions are provided for the comparison operators and code for them can be generated using `cmpPrim2`:

51a ⟨*inline handlers for comparison operators* 51a⟩≡
```
setInlineHandler("==", function(e, cb, cntxt)
   cmpPrim2(e, cb, EQ.OP, cntxt))

setInlineHandler("!=", function(e, cb, cntxt)
   cmpPrim2(e, cb, NE.OP, cntxt))

setInlineHandler("<", function(e, cb, cntxt)
   cmpPrim2(e, cb, LT.OP, cntxt))

setInlineHandler("<=", function(e, cb, cntxt)
   cmpPrim2(e, cb, LE.OP, cntxt))

setInlineHandler(">=", function(e, cb, cntxt)
   cmpPrim2(e, cb, GE.OP, cntxt))

setInlineHandler(">", function(e, cb, cntxt)
   cmpPrim2(e, cb, GT.OP, cntxt))
```

The vectorized `&` and `|` functions are handled similarly:

51b ⟨*inline handlers for* `&` *and* `|` 51b⟩≡
```
setInlineHandler("&", function(e, cb, cntxt)
   cmpPrim2(e, cb, AND.OP, cntxt))

setInlineHandler("|", function(e, cb, cntxt)
   cmpPrim2(e, cb, OR.OP, cntxt))
```

The negation operator `!` takes only one argument and code for calls to it are generated using `cmpPrim1`:

51c ⟨*inline handler for* `!` 51c⟩≡
```
setInlineHandler("!", function(e, cb, cntxt)
   cmpPrim1(e, cb, NOT.OP, cntxt))
```

11.3 Subsetting and related operations

Current R semantics are such that the subsetting operator `[` and a number of others may not evaluate some of their arguments if S3 or S4 methods are available. S-plus has different semantics—there the subsetting operator is guaranteed to evaluate its arguments. For subsetting there are `CRAN` packages that use non-standard evaluation of their arguments (`igraph` is one example), so this probably can no longer be changed.

The compiler preserve these semantics. To do so subsetting is implemented in terms of two instructions, `STARTSUBSET` and `DFLTSUBSET`. The object being subsetted is evaluated and placed on the stack. STARTSUBSET takes a constant table index for the expression and a label operand as operands and examines the object on the stack. If an internal S3 or S4 dispatch succeeds then the receiver object is removed and the result is placed on the stack and a jump to the label is carried out. If the dispatch fails then code to evaluate and execute the arguments is executed followed by a DFLTSUBSET instruction. This pattern is used for several other operations and is abstracted into the code generation function cmpDispatch. Code for subsetting and other operations is then generated by

52a ⟨*inlining handlers for some dispatching SPECIAL functions* 52a⟩≡

```
setInlineHandler("[", function(e, cb, cntxt)
    cmpDispatch(STARTSUBSET.OP, DFLTSUBSET.OP, e, cb, cntxt))

# **** c() is now a BUILTIN
# setInlineHandler("c", function(e, cb, cntxt)
#     cmpDispatch(STARTC.OP, DFLTC.OP, e, cb, cntxt, FALSE))

setInlineHandler("[[", function(e, cb, cntxt)
    cmpDispatch(STARTSUBSET2.OP, DFLTSUBSET2.OP, e, cb, cntxt))
```

The cmpDispatch function takes the two opcodes as arguments. It declines to handle cases with ... arguments in the call or with a missing first argument — these will be handled as calls to a SPECIAL primitive. For the case handled it generates code for the first argument, followed by a call to the first start.op instruction. The operands for the start.op are a constant pool index for the expression and a label for the instruction following the dflt.op instruction that allows skipping over the default case code. The default case code consists of code to compute and push the arguments followed by the dflt.op instruction.

52b ⟨cmpDispatch *function* 52b⟩≡

```
cmpDispatch <- function(start.op, dflt.op, e, cb, cntxt, missingOK = TRUE) {
    if ((missingOK && any.dots(e)) ||
        (! missingOK && dots.or.missing(e)) ||
        length(e) == 1)
        cmpSpecial(e, cb, cntxt) ## punt
    else {
        ne <- length(e)
        oe <- e[[2]]
        if (missing(oe))
            cmpSpecial(e, cb, cntxt) ## punt
        else {
            ncntxt <- make.argContext(cntxt)
            cmp(oe, cb, ncntxt)
            ci <- cb$putconst(e)
            end.label <- cb$makelabel()
            cb$putcode(start.op, ci, end.label)
            if (ne > 2)
                cmpBuiltinArgs(e[-(1:2)], names(e)[-(1:2)], cb, cntxt,
                               missingOK)
            cb$putcode(dflt.op)
            cb$putlabel(end.label)
```

```
            if (cntxt$tailcall) cb$putcode(RETURN.OP)
            TRUE
        }
    }
}
```

The `$` function is simpler to implement since its selector argument is never evaluated. The `DOLLAR` instruction takes the object to extract a component from off the stack and takes a constant index argument specifying the selection symbol.

53 ⟨*inlining handler for* `$` 53⟩≡

```
setInlineHandler("$", function(e, cb, cntxt) {
    if (any.dots(e) || length(e) != 3)
        cmpSpecial(e, cb, cntxt)
    else {
        sym <- if (is.character(e[[3]]))
            as.name(e[[3]]) else e[[3]]
        if (is.name(sym)) {
            ncntxt <- make.argContext(cntxt)
            cmp(e[[2]], cb, ncntxt)
            ci <- cb$putconst(e)
            csi <- cb$putconst(sym)
            cb$putcode(DOLLAR.OP, ci, csi)
            if (cntxt$tailcall) cb$putcode(RETURN.OP)
            TRUE
        }
        else cmpSpecial(e, cb, cntxt)
    }
})
```

11.4 Inlining simple `.Internal` functions

A number of functions are defined as simple wrappers around `.Internal` calls. One example is `dnorm`, which is currently defined as

```
dnorm <- function (x, mean = 0, sd = 1, log = FALSE)
    .Internal(dnorm(x, mean, sd, log))
```

The implementation of `.Internal` functions can be of type `BUILTIN` or `SPECIAL`. The `dnorm` implementation is of type `BUILTIN`, so its arguments are guaranteed to be evaluated in order, and this particular function doe not depend on the position of its calls in the evaluation stack. As a result, a call of the form

```
dnorm(2, 1)
```

can be replaced by the call

```
.Internal(dnorm(2, 1, 1, FALSE))
```

This can result in considerable speed-up since it avoids the overhead of the call to the wrapper function.

The substitution of a call to the wrapper with a `.Internal` call can be done by a function `inlineSimpleInternalCall` defined as

54a ⟨`inlineSimpleInternalCall` *function* 54a⟩≡
```
inlineSimpleInternalCall <- function(e, def) {
    if (! dots.or.missing(e) && is.simpleInternal(def)) {
        forms <- formals(def)
        fnames <- names(forms)
        b <- body(def)
        if (typeof(b) == "language" && length(b) == 2 && b[[1]] == "{")
            b <- b[[2]]
        icall <- b[[2]]
        defaults <- forms ## **** could strip missings but OK not to?
        cenv <- c(as.list(match.call(def, e, F))[-1], defaults)
        subst <- function(n)
            if (typeof(n) == "symbol") cenv[[as.character(n)]] else n
        args <- lapply(as.list(icall[-1]), subst)
        as.call(list(quote(.Internal), as.call(c(icall[[1]], args))))
    }
    else NULL
}
```

Code for an inlined simple internal function can then be generated by `cmpSimpleInternal`:

54b ⟨`cmpSimpleInternal` *function* 54b⟩≡
```
cmpSimpleInternal <- function(e, cb, cntxt) {
    if (any.dots(e))
        FALSE
    else {
        name <- as.character(e[[1]])
        def <- findFunDef(name, cntxt)
        if (! checkCall(def, e, NULL)) return(FALSE)
        call <- inlineSimpleInternalCall(e, def)
        if (is.null(call))
            FALSE
        else {
            cmp(call, cb, cntxt)
            TRUE
        }
    }
}
```

54c ⟨*inline safe simple* `.Internal` *functions from* `base` 54c⟩≡
```
safeBaseInternals <- c("atan2", "besselY", "beta", "choose",
                       "drop", "inherits", "is.vector", "lbeta", "lchoose",
                       "nchar", "polyroot", "typeof", "vector", "which.max",
                       "which.min", "is.loaded", "identical")

for (i in safeBaseInternals) setInlineHandler(i,  cmpSimpleInternal)
```

55a ⟨*inline safe simple* `.Internal` *functions from* `stats` 55a⟩≡
```
safeStatsInternals <- c("dbinom", "dcauchy", "dgeom", "dhyper", "dlnorm",
                        "dlogis", "dnorm", "dpois", "dunif", "dweibull",
                        "fft", "mvfft", "pbinom", "pcauchy",
                        "pgeom", "phyper", "plnorm", "plogis", "pnorm",
                        "ppois", "punif", "pweibull", "qbinom", "qcauchy",
                        "qgeom", "qhyper", "qlnorm", "qlogis", "qnorm",
                        "qpois", "qunif", "qweibull", "rbinom", "rcauchy",
                        "rgeom", "rhyper", "rlnorm", "rlogis", "rnorm",
                        "rpois", "rsignrank",  "runif", "rweibull",
                        "rwilcox", "ptukey", "qtukey")

for (i in safeStatsInternals) setInlineHandler(i,  cmpSimpleInternal, "stats")
```

It is possible to automate the process of identifying functions with the simple wrapper form and with `.Internal` implementations of type `BUILTIN`, and the function `simpleInternals` produces a list of such candidates for a given package on the search path. But determining whether such a candidate can be safely inlined needs to be done manually. Most can, but some, such as `sys.call`, cannot since they depend on their position on the call stack (removing the wrapper call that the implementation expects would change the result). Nevertheless, `simpleInternals` is useful for providing a list of candidates to screen. The `is.simpleInternal` function can be used in test code to check that the assumption made in the compiler is valid. The implementation is

55b ⟨`simpleInternals` *function* 55b⟩≡
```
simpleInternals <- function(pos = "package:base") {
    names <- ls(pos = pos, all = T)
    if (length(names) == 0)
        character(0)
    else {
        fn <-  function(n)
            is.simpleInternal(get(n, pos = pos))
        names[sapply(names, fn)]
    }
}
```

55c ⟨`is.simpleInternal` *function* 55c⟩≡
```
is.simpleInternal <- function(def) {
    if (typeof(def) == "closure" && simpleFormals(def)) {
        b <- body(def)
        if (typeof(b) == "language" && length(b) == 2 && b[[1]] == "{")
            b <- b[[2]]
        if (typeof(b) == "language" &&
            typeof(b[[1]]) == "symbol" &&
            b[[1]] == ".Internal") {
            icall <- b[[2]]
            ifun <- icall[[1]]
            typeof(ifun) == "symbol" &&
            .Internal(is.builtin.internal(as.name(ifun))) &&
            simpleArgs(icall, names(formals(def)))
```

```
        }
        else FALSE
    }
    else FALSE
}
```

56a ⟨simpleFormals *function* 56a⟩≡

```
simpleFormals <- function(def) {
    forms <- formals(def)
    if ("..." %in% names(forms))
        return(FALSE)
    for (d in as.list(forms)) {
        if (! missing(d)) {
            ## **** check constant folding
            if (typeof(d) %in% c("symbol", "language", "promise", "bytecode"))
                return(FALSE)
        }
    }
    TRUE
}
```

56b ⟨simpleArgs *function* 56b⟩≡

```
simpleArgs <- function(icall, fnames) {
    for (a in as.list(icall[-1])) {
        if (missing(a))
            return(FALSE)
        else if (typeof(a) == "symbol") {
            if (! (as.character(a) %in% fnames))
                return(FALSE)
        }
        else if (typeof(a) %in% c("language", "promise", "bytecode"))
            return(FALSE)
    }
    TRUE
}
```

11.5 Inlining `is.xyz` functions

Most of the `is.xyz` functions in `base` are simple `BUILTIN`s that do not do internal dispatch. They have simple instructions defined for them and are compiled in a common way. `cmpIs` abstract out the common compilation process.

56c ⟨cmpIs *function* 56c⟩≡

```
cmpIs <- function(op, e, cb, cntxt) {
    if (any.dots(e) || length(e) != 2)
        cmpBuiltin(e, cb, cntxt)
    else {
        ## **** check that the function is a builtin somewhere??
```

```
        s<-make.argContext(cntxt)
        cmp(e[[2]], cb, s)
        cb$putcode(op)
        if (cntxt$tailcall) cb$putcode(RETURN.OP)
        TRUE
    }
}
```

Inlining handlers are then defined by

57a ⟨*inlining handlers for* `is.xyz` *functions* 57a⟩≡

```
setInlineHandler("is.character", function(e, cb, cntxt)
    cmpIs(ISCHARACTER.OP, e, cb, cntxt))
setInlineHandler("is.complex", function(e, cb, cntxt)
    cmpIs(ISCOMPLEX.OP, e, cb, cntxt))
setInlineHandler("is.double", function(e, cb, cntxt)
    cmpIs(ISDOUBLE.OP, e, cb, cntxt))
setInlineHandler("is.integer", function(e, cb, cntxt)
    cmpIs(ISINTEGER.OP, e, cb, cntxt))
setInlineHandler("is.logical", function(e, cb, cntxt)
    cmpIs(ISLOGICAL.OP, e, cb, cntxt))
setInlineHandler("is.name", function(e, cb, cntxt)
     cmpIs(ISSYMBOL.OP, e, cb, cntxt))
setInlineHandler("is.null", function(e, cb, cntxt)
    cmpIs(ISNULL.OP, e, cb, cntxt))
setInlineHandler("is.object", function(e, cb, cntxt)
    cmpIs(ISOBJECT.OP, e, cb, cntxt))
setInlineHandler("is.symbol", function(e, cb, cntxt)
    cmpIs(ISSYMBOL.OP, e, cb, cntxt))
```

At present `is.numeric`, `is.matrix`, and `is.array` do internal dispatching so we just handle them as ordinary `BUILTIN`s. It might be worth defining virtual machine instructions for them as well.

11.6 Inlining handlers for controlling warnings

The inlining handlers in this section do not actually affect code generation. Their purpose is to suppress warnings.

Compiling calls to the `::` and `:::` functions without special handling would generate undefined variable warnings for the arguments. This is avoided by converting the arguments from symbols to strings, which these functions would do anyway at runtime, and then compiling the modified calls. The common process is handled by `cmpMultiColon`.

57b ⟨`cmpMultiColon` *function* 57b⟩≡

```
cmpMultiColon <- function(e, cb, cntxt) {
    if (! dots.or.missing(e) && length(e) == 3) {
        goodType <- function(a)
            typeof(a) == "symbol" ||
            (typeof(a) == "character" && length(a) == 1)
        fun <- e[[1]]
```

```
        x <- e[[2]]
        y <- e[[3]]
        if (goodType(x) && goodType(y)) {
            args <- list(as.character(x), as.character(y))
            cmpCallSymFun(fun, args, e, cb, cntxt)
            TRUE
        }
        else FALSE
    }
    else FALSE
}
```

Code generators are then registered by

58a ⟨*inlining handlers for* `::` *and* `:::` 58a⟩≡
```
setInlineHandler("::", cmpMultiColon)
setInlineHandler(":::", cmpMultiColon)
```

Calls to with will often generate spurious undefined variable warning for variables appearing in the expression argument. A crude approach is to compile the entire call with undefined variable warnings suppressed.

58b ⟨*inlining handler for* `with` 58b⟩≡
```
setInlineHandler("with", function(e, cb, cntxt) {
    cntxt$suppressUndefined <- TRUE
    cmpCallSymFun(e[[1]], e[-1], e, cb, cntxt)
    TRUE
})
```

A similar issue arises for `require`, where an unquoted argument is often used.

58c ⟨*inlining handler for* `require` 58c⟩≡
```
setInlineHandler("require", function(e, cb, cntxt) {
    cntxt$suppressUndefined <- TRUE
    cmpCallSymFun(e[[1]], e[-1], e, cb, cntxt)
    TRUE
})
```

12 The `switch` function

The `switch` function has somewhat awkward semantics that vary depending on whether the value of the first argument is a character string or is numeric. For a string all or all but one of the alternatives must be named, and empty case arguments are allowed and result in falling through to the next non-empty case. In the numeric case selecting an empty case produces an error. If there is more than one alternative case and no cases are named then a character selector argument will produce an error, so one can assume that a numeric switch is intended. But a `switch` with named arguments can be used with a numeric selector, so it is not in general possible to determine the intended type of the `switch` call from the structure of the call alone. The compiled code therefore has to allow for both possibilities.

The inlining handler goes through a number of steps collecting and processing information computed from the call and finally emits code for the non-empty alternatives. If the `switch` expression appears in tail position then each alternative will end in a `RETURN` instruction. If the call is not in tail position then each alternative will end with a `GOTO` than jumps to a label placed after the code for the final alternative.

59a ⟨*inline handler for* `switch` 59a⟩≡
```
setInlineHandler("switch", function(e, cb, cntxt) {
    if (length(e) < 2 || any.dots(e))
        cmpSpecial(e, cb, cntxt)
    else {
        ## **** check name on EXPR, if any, partially matches EXPR?
        ⟨extract the switch expression components 59b⟩

        ⟨collect information on named alternatives 60a⟩

        ⟨create the labels 60b⟩

        ⟨create the map from names to labels for a character switch 61a⟩

        ⟨emit code for the EXPR argument 61c⟩

        ⟨emit the switch instruction 61d⟩

        ⟨emit error code for empty alternative in numerical switch 62a⟩

        ⟨emit code for the default case 62b⟩

        ⟨emit code for non-empty alternatives 62c⟩

        if (! cntxt$tailcall)
            cb$putlabel(endLabel)
    }
    TRUE
})
```

The first step in processing the `switch` expression is to extract the selector expression `expr` and the case expressions, to identify which, if any, of the cases are empty, and to extract the names of the cases as `nm`. If there is only one case and that case is not named then setting `nm = ""` allows this situation to be processed by code used when names are present.

59b ⟨*extract the* `switch` *expression components* 59b⟩≡
```
expr <- e[[2]]
cases <-e[-c(1, 2)]

miss <- missingArgs(cases)
nm <- names(cases)

## allow for corner cases like switch(x, 1) which always
## returns 1 if x is a character scalar.
```

```
if (is.null(nm) && length(cases) == 1)
    nm <- ""
```

The next step in the case where some cases are named is to check for a default expression. If there is more than one expression then the `switch` is compiled by `cmpSpecial`. This avoids having to reproduce the runtime error that would be generated if the `switch` is called with a character selector.

60a ⟨*collect information on named alternatives* 60a⟩≡

```
## collect information on named alternatives and check for
## multiple default cases.
if (! is.null(nm)) {
    haveNames <- TRUE
    ndflt <- sum(nm == "")
    if (ndflt > 1) {
        notifyMultipleSwitchDefaults(ndflt, cntxt)
        ## **** punt back to interpreted version for now to get
        ## **** runtime error message for multiple defaults
        cmpSpecial(e, cb, cntxt)
        return(TRUE)
    }
    if (ndflt > 0)
        haveCharDflt <- TRUE
    else
        haveCharDflt <- FALSE
}
else {
    haveNames <- FALSE
    haveCharDflt <- FALSE
}
```

Next the labels are generated. `missLabel` will be the label for code that signals an error if a numerical selector expression chooses a case with an empty argument. The label `dfltLabel` will be for code that invisibly procures the value `NULL`, which is the default case for a numerical selector argument and also for a character selector when no unnamed default case is provided. All non-empty cases are given their own labels, and `endLabel` is generated if it will be needed as the `GOTO` target for a `switch` expression that is not in tail position.

60b ⟨*create the labels* 60b⟩≡

```
## create the labels
if (any(miss))
    missLabel <- cb$makelabel()
dfltLabel <- cb$makelabel()

lab <- function(m)
    if (m) missLabel
    else cb$makelabel()
labels <- c(lapply(miss, lab), list(dfltLabel))

if (! cntxt$tailcall)
```

```
    endLabel <- cb$makelabel()
```

When there are named cases a map from the case names to the corresponding code labels is constructed next. If no unnamed default was provided one is added that uses the `dfltLabel`.

61a ⟨*create the map from names to labels for a character switch* 61a⟩≡

```
## create the map from names to labels for a character switch
if (haveNames) {
    unm <- unique(nm[nm != ""])
    if (haveCharDflt)
        unm <- c(unm, "")
    nlabels <- labels[unlist(lapply(unm, findActionIndex, nm, miss))]
    ## if there is no unnamed case to act as a default for a
    ## character switch then the numeric default becomes the
    ## character default as well.
    if (! haveCharDflt) {
        unm <- c(unm, "")
        nlabels <- c(nlabels, list(dfltLabel))
    }
}
else {
    unm <- NULL
    nlabels <- NULL
}
```

The computation of the index of the appropriate label for a given name is carried out by `findActionIndex`.

61b ⟨`findActionIndex` *function* 61b⟩≡

```
findActionIndex <- function(name, nm, miss) {
    start <- match(name, nm)
    aidx <- c(which(! miss), length(nm) + 1)
    min(aidx[aidx >= start])
}
```

At this point we are ready to start emitting code into the code buffer. First code to compute the selector is emitted. As with the condition for an `if` expression a non-tail-call context is used.

61c ⟨*emit code for the* `EXPR` *argument* 61c⟩≡

```
## compile the EXPR argument
ncntxt <- make.nonTailCallContext(cntxt)
cmp(expr, cb, ncntxt)
```

The switch instruction takes the selector off the stack and four operands form the instruction stream: the call index, an index for the names, or `NULL` if there are none, and indices for the labels for a character selector and for a numeric selector. At this point lists of labels are placed in the instruction buffer. At code extraction time these will be replaced by indices for numeric offset vectors by the `patchlables` function of the code buffer.

61d ⟨*emit the switch instruction* 61d⟩≡

```
## emit the SWITCH instruction
cei <- cb$putconst(e)
if (haveNames) {
    cni <- cb$putconst(unm)
```

```
        cb$putcode(SWITCH.OP, cei, cni, nlabels, labels)
    }
    else {
        cni <- cb$putconst(NULL)
        cb$putcode(SWITCH.OP, cei, cni, cni, labels)
    }
```

If there are empty alternatives then code to signal an error for a numeric selector that chooses one of these is needed and is identified by the label `missLabel`.

62a ⟨*emit error code for empty alternative in numerical switch* 62a⟩≡

```
## emit code to signal an error if a numeric switch hist an
## empty alternative (fall through, as for character, might
## make more sense but that isn't the way switch() works)
if (any(miss)) {
    cb$putlabel(missLabel)
    cmp(quote(stop("empty alternative in numeric switch")), cb, cntxt)
}
```

Code for the numeric default case, corresponding to `dfltLabel`, places `NULL` on the stack, and for a `switch` in tail position this is followed by an `INVISIBLE` and a `RETURN` instruction.

62b ⟨*emit code for the default case* 62b⟩≡

```
## emit code for the default case
cb$putlabel(dfltLabel)
cb$putcode(LDNULL.OP)
if (cntxt$tailcall) {
    cb$putcode(INVISIBLE.OP)
    cb$putcode(RETURN.OP)
}
else
    cb$putcode(GOTO.OP, endLabel)
```

Finally the labels and code for the non-empty alternatives are written to the code buffer. In non-tail position the code is followed by a `GOTO` instruction that jumps to `endLabel`. The final case does not need this `GOTO`.

62c ⟨*emit code for non-empty alternatives* 62c⟩≡

```
## emit code for the non-empty alternatives
for (i in seq_along(cases)) {
    if (! miss[i]) {
        cb$putlabel(labels[[i]])
        cmp(cases[[i]], cb, cntxt)
        if (! cntxt$tailcall)
            cb$putcode(GOTO.OP, endLabel)
    }
}
```

13 Assignments expressions

R supports simple assignments in which the left-hand side of the assignment expression is a symbol and complex assignments of the form

```
f(x) <- v
```

or

```
g(f(x)) <- v
```

The second form is sometimes called a nested complex assignment. Ordinary assignment creates or modifies a binding in the current environment. Superassignment via the `<<-` operator modifies a binding in a containing environment.

Assignment expressions are compiled by `cmpAssign`. This function checks the form of the assignment expression and, for well formed expressions then uses `cmpSymbolAssign` for simple assignments and `cmpComplexAssign` for complex assignments.

63a ⟨`cmpAssign` *function* 63a⟩≡
```
cmpAssign <- function(e, cb, cntxt) {
    if (! checkAssign(e, cntxt))
        return(cmpSpecial(e, cb, cntxt))
    superAssign <- as.character(e[[1]]) == "<<-"
    lhs <- e[[2]]
    value <- e[[3]]
    symbol <- as.name(getAssignedVar(e))
    if (superAssign && ! findVar(symbol, cntxt))
        notifyNoSuperAssignVar(symbol, cntxt)
    if (is.name(lhs) || is.character(lhs))
        cmpSymbolAssign(symbol, value, superAssign, cb, cntxt)
    else if (typeof(lhs) == "language")
        cmpComplexAssign(symbol, lhs, value, superAssign, cb, cntxt)
    else cmpSpecial(e, cb, cntxt) # punt for now
}
```

The code generators for the assignment operators `<-` and `=` and the superassignment operator `<<-` are registered by

63b ⟨*inlining handlers for* `<-`, `=`, *and* `<<-` 63b⟩≡
```
setInlineHandler("<-", cmpAssign)
setInlineHandler("=", cmpAssign)
setInlineHandler("<<-", cmpAssign)
```

The function `checkAssign` is used to check that an assignment expression is well-formed.

63c ⟨`checkAssign` *function* 63c⟩≡
```
checkAssign <- function(e, cntxt) {
    if (length(e) != 3)
        FALSE
    else {
        place <- e[[2]]
        if (typeof(place) == "symbol" ||
```

```
            (typeof(place) == "character" && length(place) == 1))
            TRUE
        else {
            ⟨check left hand side call 64a⟩
        }
    }
}
```

A valid left hand side call must have a function that is either a symbol or is of the form `foo::bar` or `foo:::bar`, and the first argument must be a symbol or another valid left hand side call. A `while` loop is used to unravel nested calls.

64a ⟨*check left hand side call* 64a⟩≡

```
while (typeof(place) == "language") {
    fun <- place[[1]]
    if (typeof(fun) != "symbol" &&
        ! (typeof(fun) == "language" && length(fun) == 3 &&
           typeof(fun[[1]]) == "symbol" &&
           as.character(fun[[1]]) %in% c("::", ":::"))) {
        notifyBadAssignFun(fun, cntxt)
        return(FALSE)
    }
    place = place[[2]]
}
if (typeof(place) == "symbol")
    TRUE
else FALSE
```

13.1 Simple assignment expressions

Code for assignment to a symbol is generated by `cmpSymbolAssign`.

64b ⟨`cmpSymbolAssign` *function* 64b⟩≡

```
cmpSymbolAssign <- function(symbol, value, superAssign, cb, cntxt) {
    ⟨compile the right hand side value expression 64c⟩
    ⟨emit code for the symbol assignment instruction 64d⟩
    ⟨for tail calls return the value invisibly 65a⟩
    TRUE
}
```

A non-tail-call context is used to generate code for the right hand side value expression.

64c ⟨*compile the right hand side value expression* 64c⟩≡

```
ncntxt <- make.nonTailCallContext(cntxt)
cmp(value, cb, ncntxt)
```

The `SETVAR` and `SETVAR2` instructions assign the value on the stack to the symbol specified by its constant pool index operand. The `SETVAR` instruction is used by ordinary assignment to assign in the local frame, and `SETVAR2` for superassignments.

64d ⟨*emit code for the symbol assignment instruction* 64d⟩≡

```
ci <- cb$putconst(symbol)
```

```
if (superAssign)
    cb$putcode(SETVAR2.OP, ci)
else
    cb$putcode(SETVAR.OP, ci)
```

The super-assignment case does not need to check for and warn about a missing binding since this is done in `cmpAssign`.

The `SETVAR` and `SETVAR2` instructions leave the value on the stack as the value of the assignment expression; if the expression appears in tail position then this value is returned with the visible flag set to `FALSE`.

65a ⟨*for tail calls return the value invisibly* 65a⟩≡

```
if (cntxt$tailcall) {
    cb$putcode(INVISIBLE.OP)
    cb$putcode(RETURN.OP)
}
```

13.2 Complex assignment expressions

It seems somehow appropriate at this point to mention that the code in `eval.c` implementing the interpreter semantics starts with the following comment:

```
/*
 *  Assignments for complex LVAL specifications. This is the stuff that
 *  nightmares are made of ...
```

There are some issues with the semantics for complex assignment as implemented by the interpreter:

- With the current approach the following legal, though strange, code fails:

  65b ⟨*inner assignment trashes temporary* 65b⟩≡

  ```
  f <-function(x, y) x
  `f<-` <- function(x, y, value) { y; x}
  x <- 1
  y <- 2
  f(x, y[] <- 1) <- 3
  ```

  The reason is that the current left hand side object is maintained in a variable `*tmp*`, and processing the assignment in the second argument first overwrites the value of `*tmp*` and then removes `*tmp*` before the first argument is evaluated. Using evaluated promises as arguments, as is done for the right hand side value, solves this.

- The current approach of using a temporary variable `*tmp*` to hold the evaluated LHS object requires an internal cleanup context to ensure that the variable is removed in the event of a non-local exit. Implementing this in the compiler would introduce significant overhead.

- The asymmetry of handling the pre-evaluated right hand side value via an evaluated promise and the pre-evaluated left hand side via a temporary variable makes the code harder to understand and the semantics harder to explain.

- Using promises in an expression passed to eval means promises can leak out into R via sys.call. This is something we have tried to avoid and should try to avoid so we can have the freedom to implement lazy evaluation differently if that seems useful. [It may be possible at some point to avoid allocation of promise objects in compiled code.] The compiler can avoid this by using promises only in the argument lists passed to function calls, not in the call expressions. A similar change could be made in the interpreter but it would have a small runtime penalty for constructing an expression in addition to an argument list I would prefer to avoid that for now until the compiler has been turned on by default.
- The current approach of installing the intermediate RHS value as the expression for the RHS promise in nested complex assignments has several drawbacks:
  - it can produce huge expressions.
  - the result is misleading if the intermediate RHS value is a symbol or a language object.
  - to maintain this in compiled code it would be necessary to construct the assignment function call expression at runtime even though it is usually not needed (or it would require significant rewriting to allow on-demand computation of the call). If *vtmp* is used as a marker for the expression and documented as not a real variable then the call can be constructed at compile time.
- In nested complex assignments the additional arguments of the inner functions are evaluated twice. This is illustrated by running this code:

66 ⟨*multiple evaluation of arguments in assignments* 66⟩≡

```
f <- function(x, y) {y ; x }
`f<-` <- function(x, y, value) { y; x }
g <- function(x, y) {y ; x }
`g<-` <- function(x, y, value) { y; x }
x <- 1
y <- 2
f(g(x, print(y)), y) <- 3
```

  This is something we have lived with, and I don't propose to change it at this time. But it would be good to be able to change it in the future.

Because of these issues the compiler implements slightly different semantics for complex assignment than the current intepreter. *Evaluation* semantics should be identical; the difference arises in how intermediate values are managed and has some effect on results produced by `substitute`. In particular, no intermediate `*tmp*` value is used and therefore no cleanup frame is needed. This does mean that uses of the form

```
eval(substitute(<first arg>), parent.frame())
```

will no longer work. In tests of most of CRAN and BioC this directly affected only one function, `$.proto` in the `proto` package, and indirectly about 30 packages using proto failed. I looked at the `$.proto` implementation, and it turned out that the `eval(substitute())` approach used there could be replaced by standard evaluation using lexical scope. This produces better code, and the result works with both the current R interpreter and compiled code (proto and all the dependent packages pass check with this change). The `proto` maintainer has changed `proto` along these lines. It would be good to soon change the interpreter to also use evaluated promises in place of the `*tmp*` variable to bring the compiled and interpreted semantics closer together.

Complex assignment expressions are compiled by `cmpComplexAssign`.

67a ⟨`cmpComplexAssign` *function* 67a⟩≡
```
cmpComplexAssign <- function(symbol, lhs, value, superAssign, cb, cntxt) {
    ⟨select complex assignment instructions 67c⟩
    ⟨compile the right hand side value expression 64c⟩
    ⟨compile the left hand side call 67b⟩
    ⟨for tail calls return the value invisibly 65a⟩
    TRUE;
}
```

Assignment code is bracketed by a start and an end instruction.

67b ⟨*compile the left hand side call* 67b⟩≡
```
csi <- cb$putconst(symbol)
cb$putcode(startOP, csi)

⟨compile code to compute left hand side values 68a⟩
⟨compile code to compute right hand side values 69a⟩

cb$putcode(endOP, csi)
```

The appropriate instructions `startOP` and `endOP` depend on whether the assignment is an ordinary assignment or a superassignment.

67c ⟨*select complex assignment instructions* 67c⟩≡
```
if (superAssign) {
    startOP <- STARTASSIGN2.OP
    endOP <- ENDASSIGN2.OP
}
else {
    if (! findVar(symbol, cntxt))
        notifyUndefVar(symbol, cntxt)
    startOP <- STARTASSIGN.OP
    endOP <- ENDASSIGN.OP
}
```

An undefined variable notification is issued for ordinary assignment, since this will produce a runtime error. For superassignment `cmpAssign` has already checked for an undefined left-hand-side variable and issued a notification if none was found.

The start instructions obtain the initial value of the left-hand-side variable and in the case of standard assignment assign it in the local frame if it is not assigned there already. They also prepare the stack for the assignment process. The stack invariant maintained by the assignment process is that the current right hand side value is on the top, followed by the evaluated left hand side values and the original right hand side value. Thus the start instruction leaves the right hand side value, the value of the left hand side variable, and again the right hand side value on the top of the stack.

The end instruction finds the final right hand side value followed by the original right hand side value on the top of the stack. The final value is removed and assigned to the appropriate variable binding. The original right hand side value is left on the top of the stack as the value of the assignment expression.

Evaluating a nested complex assignment involves evaluating a sequence of expressions to obtain the left hand sides to modify, and then evaluating a sequence of corresponding calls to replacement functions in the opposite order. The function `flattenPlace` returns a list of the expressions that need to be considered, with `*tmp*` in place of the current left hand side argument. For example, for an assignment of the form `f(g(h(x, k), j), i) <- v` this produces

```
> flattenPlace(quote(f(g(h(x, k), j), i)))
{\tt{}1}
f(`*tmp*`, i)

{\tt{}2}
g(`*tmp*`, j)

{\tt{}3}
h(`*tmp*`, k)
```

The sequence of left hand side values needed consists of the original variable value, which is already on the stack, and the values of `` h(`*tmp*`, k) `` and `` g(`*tmp*`, j) ``.

In general the additional evaluations needed are of all but the first expression produced by `flattenPlace`, evaluated in reverse order. An argument context is used since there are already values on the stack.

68a ⟨*compile code to compute left hand side values* 68a⟩≡

```
ncntxt <- make.argContext(cntxt)
flatPlace <- flattenPlace(lhs)
for (p in rev(flatPlace[-1]))
    cmpGetterCall(p, cb, ncntxt)
```

The compilation of the individual calls carried out by `cmpGetterCall`, which is presented in Section 13.4. Each compilation places the new left hand side value on the top of the stack and then switches it with the value below, which is the original right hand side value, to preserve the stack invariant.

The function `flattenPlace` is defined as

68b ⟨`flattenPlace` *function* 68b⟩≡

```
flattenPlace <- function(place) {
    places <- NULL
    while (typeof(place) == "language") {
        if (length(place) < 2)
            stop("bad assignment 1")
        tplace <- place
        tplace[[2]] <- as.name("*tmp*")
        places <- c(places, list(tplace))
        place <- place[[2]]
    }
    if (typeof(place) != "symbol")
        stop("bad assignment 2")
```

```
    places
}
```

After the right hand side values have been computed the stack contains the original right hand side value followed by the left hand side values in the order in which they need to be modified. Code to call the sequence of replacement functions is generated by

69a ⟨*compile code to compute right hand side values* 69a⟩≡
```
cmpSetterCall(flatPlace[[1]], value, cb, ncntxt)
for (p in flatPlace[-1])
    cmpSetterCall(p, as.name("*vtmp*"), cb, ncntxt)
```

The first call uses the expression for the original right hand side in its call; all others will use `*vtmp*`. Each replacement function call compiled by `cmpSetterCall` will remove the top two elements from the stack and then push the new right hand side value on the stack. `cmpSetterCall` is described in Section 13.3.

13.3 Compiling setter calls

Setter calls, or calls to replacement functions, in compiled assignment expressions find stack that contains the current right hand side value on the top followed by the current left hand side value. Some replacement function calls, such as calls to `$<-`, are handled by an inlining mechanism described below. The general case when the function is specified by a symbol is handled a `GETFUN` instruction to push the function on the stack, pushing any additional arguments on the stack, and using the `SETTER_CALL` instruction to execute the call. This instruction adjusts the argument list by inserting as the first argument an evaluated promise for the left hand side value and as the last argument an evaluated promise for the right hand side value; the final argument also has the `value` tag. The case where the function is specified in the form `foo::bar` or `foo:::bar` differs only compiling the function expression and using `CHECKFUN` to verify the result and prepare the stack.

69b ⟨`cmpSetterCall` *function* 69b⟩≡
```
cmpSetterCall <- function(place, vexpr, cb, cntxt) {
    afun <- getAssignFun(place[[1]])
    acall <- as.call(c(afun, as.list(place[-1]), list(value = vexpr)))
    acall[[2]] <- as.name("*tmp*")
    ncntxt <- make.callContext(cntxt, acall)
    if (is.null(afun))
        ## **** warn instead and arrange for cmpSpecial?
        ## **** or generate code to signal runtime error?
        cntxt$stop(gettext("invalid function in complex assignment"))
    else if (typeof(afun) == "symbol") {
        if (! trySetterInline(afun, place, acall, cb, ncntxt)) {
            ci <- cb$putconst(afun)
            cb$putcode(GETFUN.OP, ci)
            ⟨compile additional arguments and call to setter function 70a⟩
        }
    }
    else {
        cmp(afun, cb, ncntxt)
```

```
        cb$putcode(CHECKFUN.OP)
        ⟨compile additional arguments and call to setter function 70a⟩
    }
}
```

The common code for compiling additional arguments and issuing the SETTER_CALL instruction is given by

70a ⟨*compile additional arguments and call to setter function* 70a⟩≡

```
cb$putcode(PUSHNULLARG.OP)
cmpCallArgs(place[-c(1, 2)], cb, ncntxt)
cci <- cb$putconst(acall)
cvi <- cb$putconst(vexpr)
cb$putcode(SETTER_CALL.OP, cci, cvi)
```

The PUSHNULL instruction places NULL in the argument list as a first argument to serve as a place holder; SETTER_CALL replaces this with the evaluated promise for the current left hand side value.

The replacement function corresponding to fun is computed by getAssignFun. If fun is a symbol then the assignment function is the symbol followed by <-. The function fun can also be an expression of the form foo::bar, in which case the replacement function is the expression foo::‘bar<-‘. NULL is returned if fun does not fit into one of these two cases.

70b ⟨getAssignFun *function* 70b⟩≡

```
getAssignFun <- function(fun) {
    if (typeof(fun) == "symbol")
        as.name(paste0(fun, "<-"))
    else {
        ## check for and handle foo::bar(x) <- y assignments here
        if (typeof(fun) == "language" && length(fun) == 3 &&
            (as.character(fun[[1]]) %in% c("::", ":::")) &&
            typeof(fun[[2]]) == "symbol" && typeof(fun[[3]]) == "symbol") {
            afun <- fun
            afun[[3]] <- as.name(paste0(fun[[3]],"<-"))
            afun
        }
        else NULL
    }
}
```

To produce more efficient code some replacement function calls can be inlined and use specialized instructions. The most important of these are $<-, [<-, and [[<-. An inlining mechanism similar to the one described in Section 6 is used for this purpose. A separate mechanism is needed because of the fact that in the present context two arguments, the left hand side and right hand side values, are already on the stack.

70c ⟨*setter inlining mechanism* 70c⟩≡

```
setterInlineHandlers <- new.env(hash = TRUE, parent = emptyenv())

setSetterInlineHandler <- function(name, h, package = "base") {
    if (exists(name, setterInlineHandlers, inherits = FALSE)) {
        entry <- get(name, setterInlineHandlers)
```

```
        if (entry$package != package) {
            fmt <- "handler for '%s' is already defined for another package"
            stop(gettextf(fmt, name), domain = NA)
        }
    }
    entry <- list(handler = h, package = package)
    assign(name, entry, setterInlineHandlers)
}

getSetterInlineHandler <- function(name, package = "base") {
    if (exists(name, setterInlineHandlers, inherits = FALSE)) {
        hinfo <- get(name, setterInlineHandlers)
        if (hinfo$package == package)
            hinfo$handler
        else NULL
    }
    else NULL
}

trySetterInline <- function(afun, place, call, cb, cntxt) {
    name <- as.character(afun)
    info <- getInlineInfo(name, cntxt)
    if (is.null(info))
        FALSE
    else {
        h <- getSetterInlineHandler(name, info$package)
        if (! is.null(h))
            h(afun, place, call, cb, cntxt)
        else FALSE
    }
}
```

The inline handler for `$<-` replacement calls uses the `DOLLARGETS` instruction. The handler declines to handle cases that would produce runtime errors; these are compiled by the generic mechanism.

71 ⟨*setter inline handler for* `$<-` 71⟩≡

```
setSetterInlineHandler("$<-", function(afun, place, call, cb, cntxt) {
    if (any.dots(place) || length(place) != 3)
        FALSE
    else {
        sym <- place[[3]]
        if (is.character(sym))
            sym <- as.name(sym)
        if (is.name(sym)) {
            ci <- cb$putconst(call)
            csi <- cb$putconst(sym)
            cb$putcode(DOLLARGETS.OP, ci, csi)
            TRUE
```

```
        }
        else FALSE
    }
})
```

The replacement functions `[<-` and `[[<-]` are implemented as `SPECIAL` functions that do internal dispatching. They are therefore compiled along the same lines as their corresponding accessor functions as described in Section 11.3. The common pattern is implemented by `cmpSetterDispatch`.

72a ⟨`cmpSetterDispatch` *function* 72a⟩≡

```
cmpSetterDispatch <- function(start.op, dflt.op, afun, place, call, cb, cntxt) {
    if (any.dots(place))
        FALSE ## punt
    else {
        ci <- cb$putconst(call)
        end.label <- cb$makelabel()
        cb$putcode(start.op, ci, end.label)
        if (length(place) > 2) {
            args <- place[-(1:2)]
            cmpBuiltinArgs(args, names(args), cb, cntxt, TRUE)
        }
        cb$putcode(dflt.op)
        cb$putlabel(end.label)
        TRUE
    }
}
```

The two inlining handlers are then defined as

72b ⟨*setter inline handlers for* `[<-` *and* `[[<-` 72b⟩≡

```
setSetterInlineHandler("[<-", function(afun, place, call, cb, cntxt)
    cmpSetterDispatch(STARTSUBASSIGN.OP, DFLTSUBASSIGN.OP,
                      afun, place, call, cb, cntxt))

setSetterInlineHandler("[[<-", function(afun, place, call, cb, cntxt)
    cmpSetterDispatch(STARTSUBASSIGN2.OP, DFLTSUBASSIGN2.OP,
                      afun, place, call, cb, cntxt))
```

An inline handler is defined for `@<-` in order to suppress spurious warnings about the slot name symbol. A call in which the slot is specified by a symbol is converted to one using a string instead, and is then compiled by a recursive call to `cmpSetterCall`; the handler will decline in this second call and the default compilation strategy will be used.

72c ⟨*setter inlining handler for* `@<-` 72c⟩≡

```
setSetterInlineHandler("@<-", function(afun, place, acall, cb, cntxt) {
    if (! dots.or.missing(place) && length(place) == 3 &&
        typeof(place[[3]]) == "symbol") {
        place[[3]] <- as.character(place[[3]])
        vexpr <- acall[[length(acall)]]
        cmpSetterCall(place, vexpr, cb, cntxt)
        TRUE
```

```
    }
    else FALSE
}, "methods")
```

13.4 Compiling getter calls

Getter calls within an assignment also need special handling because of the left hand side argument being on the stack already and because of the need to restore the stack invariant. There are again two cases for installing the getter function on the stack. These are then followed by common code for handling the additional arguments and the call.

73a ⟨cmpGetterCall *function* 73a⟩≡

```
cmpGetterCall <- function(place, cb, cntxt) {
    ncntxt <- make.callContext(cntxt, place)
    fun <- place[[1]]
    if (typeof(fun) == "symbol") {
        if (! tryGetterInline(place, cb, ncntxt)) {
            ci <- cb$putconst(fun)
            cb$putcode(GETFUN.OP, ci)
            ⟨compile additional arguments and call to getter function 73b⟩
        }
    }
    else {
        cmp(fun, cb, ncntxt)
        cb$putcode(CHECKFUN.OP)
        ⟨compile additional arguments and call to getter function 73b⟩
    }
}
```

In the common code, as in setter calls a `NULL` is placed on the argument stack as a place holder for the left hand side promise. Then the additional arguments are placed on the stack and the `GETTER-CALL` instruction is issued. This instruction installs the evaluated promise with the left hand side value as the first argument and executes the call. The call will leave the next right left hand side on the top of the stack. A `SWAP` instruction then switches the top two stack entries. This leaves the original right hand side value on top followed by the new left hand side value returned by the getter call and any other left hand side values produced by earlier getter call.

73b ⟨*compile additional arguments and call to getter function* 73b⟩≡

```
cb$putcode(PUSHNULLARG.OP)
cmpCallArgs(place[-c(1, 2)], cb, ncntxt)
cci <- cb$putconst(place)
cb$putcode(GETTER_CALL.OP, cci)
cb$putcode(SWAP.OP)
```

Again an inlining mechanism is needed to handle calls to functions like `$` and `[`. These are able to use the same instructions as the inline handlers in Section 11.3 for ordinary calls to `$` and `[` but require some additional work to deal with maintaining the stack invariant.

The inlining mechanism itself is analogous to the general one and the one for inlining setter calls.

74a ⟨*getter inlining mechanism* 74a⟩≡
```
getterInlineHandlers <- new.env(hash = TRUE, parent = emptyenv())

setGetterInlineHandler <- function(name, h, package = "base") {
    if (exists(name, getterInlineHandlers, inherits = FALSE)) {
        entry <- get(name, getterInlineHandlers)
        if (entry$package != package) {
            fmt <- "handler for '%s' is already defined for another package"
            stop(gettextf(fmt, name), domain = NA)
        }
    }
    entry <- list(handler = h, package = package)
    assign(name, entry, getterInlineHandlers)
}

getGetterInlineHandler <- function(name, package = "base") {
    if (exists(name, getterInlineHandlers, inherits = FALSE)) {
        hinfo <- get(name, getterInlineHandlers)
        if (hinfo$package == package)
            hinfo$handler
        else NULL
    }
    else NULL
}

tryGetterInline <- function(call, cb, cntxt) {
    name <- as.character(call[[1]])
    info <- getInlineInfo(name, cntxt)
    if (is.null(info))
        FALSE
    else {
        h <- getGetterInlineHandler(name, info$package)
        if (! is.null(h))
            h(call, cb, cntxt)
        else FALSE
    }
}
```

The inline handler for `$` in a getter context uses the `DUP2ND` instruction to push the second value on the stack, the previous left hand side value, onto the stack. The `DOLLAR` instruction pops this value, computes the component for this value and the symbol in the constant pool, and pushes the result on the stack. A `SWAP` instruction then interchanges this value with the next value, which is the original right hand side value, thus restoring the stack invariant.

74b ⟨*getter inline handler for* `$` 74b⟩≡
```
setGetterInlineHandler("$", function(call, cb, cntxt) {
    if (any.dots(call) || length(call) != 3)
        FALSE
    else {
```

```
        sym <- call[[3]]
        if (is.character(sym))
            sym <- as.name(sym)
        if (is.name(sym)) {
            ci <- cb$putconst(call)
            csi <- cb$putconst(sym)
            cb$putcode(DUP2ND.OP)
            cb$putcode(DOLLAR.OP, ci, csi)
            cb$putcode(SWAP.OP)
            TRUE
        }
        else FALSE
    }
})
```

Calls to `[` and `[[` again need two instructions to support the internal dispatch. The general pattern is implemented in `cmpGetterDispatch`. A `DUP2ND` instruction is used to place the first argument for the call on top of the stack, code analogous to the code for ordinary calls to `[` and `[[` is used to make the call, and this is followed by a `SWAP` instruction to rearrange the stack.

75a ⟨`cmpGetterDispatch` *function* 75a⟩≡

```
cmpGetterDispatch <- function(start.op, dflt.op, call, cb, cntxt) {
    if (any.dots(call))
        FALSE ## punt
    else {
        ci <- cb$putconst(call)
        end.label <- cb$makelabel()
        cb$putcode(DUP2ND.OP)
        cb$putcode(start.op, ci, end.label)
        if (length(call) > 2) {
            args <- call[-(1:2)]
            cmpBuiltinArgs(args, names(args), cb, cntxt, TRUE)
        }
        cb$putcode(dflt.op)
        cb$putlabel(end.label)
        cb$putcode(SWAP.OP)
        TRUE
    }
}
```

The two inline handlers are then defined as

75b ⟨*getter inline handlers for* `[` *and* `[[` 75b⟩≡

```
setGetterInlineHandler("[", function(call, cb, cntxt)
    cmpGetterDispatch(STARTSUBSET.OP, DFLTSUBSET.OP, call, cb, cntxt))

setGetterInlineHandler("[[", function(call, cb, cntxt)
    cmpGetterDispatch(STARTSUBSET2.OP, DFLTSUBSET2.OP, call, cb, cntxt))
```

14 Constant folding

A very valuable compiler optimization is constant folding. For example, an expression for computing a normal density function may include the code

```
1 / sqrt(2 * pi)
```

The interpreter would have to evaluate this expression each time it is needed, but a compiler can often compute the value once at compile time.

The constant folding optimization can be applied at various points in the compilation process: It can be applied to the source code before code generation or to the generated code in a separate optimization phase. For now, constant folding is applied during the code generation phase.

The `constantFold` function examines its expression argument and handles each expression type by calling an appropriate function.

76a ⟨`constantFold` *function* 76a⟩≡

```
## **** rewrite using switch??
constantFold <- function(e, cntxt) {
    type = typeof(e)
    if (type == "language")
        constantFoldCall(e, cntxt)
    else if (type == "symbol")
        constantFoldSym(e, cntxt)
    else if (type == "promise")
        cntxt$stop(gettext("cannot constant fold literal promises"),
                   cntxt)
    else if (type == "bytecode")
        cntxt$stop(gettext("cannot constant fold literal bytecode objects"),
                   cntxt)
    else checkConst(e)
}
```

The `checkConst` function decides whether a value is a constant that is small enough and simple enough to enter into the constant pool. If so, then `checkConst` wraps the value in a list as the `value` component. If not, then `NULL` is returned.

76b ⟨`checkConst` *function* 76b⟩≡

```
checkConst <- function(e) {
    if (mode(e) %in% constModes && length(e) <= maxConstSize)
        list(value = e)
    else
        NULL
}
```

The maximal size and acceptable modes are defined by

76c ⟨`maxConstSize` *and* `constModes` *definitions* 76c⟩≡

```
maxConstSize <- 10

constModes <- c("numeric", "logical", "NULL", "complex", "character")
```

For now, constant folding is only applied for a particular set of variables and functions defined in the base package. The constant folding code uses `isBaseVar` to determine whether a variable can be assumed to reference the corresponding base variable given the current compilation environment and optimization setting. `constantFoldSym` is applied to base variables in the `constNames` list.

77a ⟨`constantFoldSym` *function* 77a⟩≡
```
## Assumes all constants will be defined in base.
## Eventually allow other packages to define constants.
## Any variable with locked binding could be used if type is right.
## Allow local declaration of optimize, notinline declaration.
constantFoldSym <- function(var, cntxt) {
    var <- as.character(var)
    if (var %in% constNames && isBaseVar(var, cntxt))
        checkConst(get(var, .BaseNamespaceEnv))
    else NULL
}
```

77b ⟨`constNames` *definition* 77b⟩≡
```
constNames <- c("pi", "T", "F")
```

Call expressions are handled by determining whether the function called is eligible for constant folding, attempting to constant fold the arguments, and calling the folding function. The result is the passed to `checkConst`. If an error occurs in the call to the folding function then `constantFoldCall` returns `NULL`.

77c ⟨`constantFoldCall` *function* 77c⟩≡
```
constantFoldCall <- function(e, cntxt) {
    fun <- e[[1]]
    if (typeof(fun) == "symbol") {
        ffun <- getFoldFun(fun, cntxt)
        if (! is.null(ffun)) {
            args <- as.list(e[-1])
            for (i in seq_along(args)) {
                a <- args[[i]]
                if (missing(a))
                    return(NULL)
                val <- constantFold(a, cntxt)
                if (! is.null(val))
                    args[i] <- list(val$value) ## **** in case value is NULL
                else return(NULL)
            }
            modes <- unlist(lapply(args, mode))
            if (all(modes %in% constModes)) {
                tryCatch(checkConst(do.call(ffun, args)),
                         error = function(e) NULL) ## **** issue warning??
            }
            else NULL
        }
        else NULL
    }
```

```
    else NULL
}
```

The functions in the base package eligible for constant folding are

78a ⟨foldFuns *definition* 78a⟩≡
```
foldFuns <- c("+", "-", "*", "/", "^", "(",
              ">", ">=", "==", "!=", "<", "<=", "||", "&&", "!",
              "|", "&", "%%",
              "c", "rep", ":",
              "abs", "acos", "acosh", "asin", "asinh", "atan", "atan2",
              "atanh", "ceiling", "choose", "cos", "cosh", "exp", "expm1",
              "floor", "gamma", "lbeta", "lchoose", "lgamma", "log", "log10",
              "log1p", "log2", "max", "min", "prod", "range", "round",
              "seq_along", "seq.int", "seq_len", "sign", "signif",
              "sin", "sinh", "sqrt", "sum", "tan", "tanh", "trunc",
              "baseenv", "emptyenv", "globalenv",
              "Arg", "Conj", "Im", "Mod", "Re",
              "is.R")
```

`getFoldFun` checks the called function against this list and whether the binding for the variable can be assumed to be from the base package. If then returns the appropriate function from the base package or `NULL`.

78b ⟨getFoldFun *function* 78b⟩≡
```
## For now assume all foldable functions are in base
getFoldFun <- function(var, cntxt) {
    var <- as.character(var)
    if (var %in% foldFuns && isBaseVar(var, cntxt)) {
        val <- get(var, .BaseNamespaceEnv)
        if (is.function(val))
            val
        else
            NULL
    }
    else NULL
}
```

15 More top level functions

15.1 Compiling closures

The function `cmpfun` is for compiling a closure. The body is compiled with `genCode` and combined with the closure's formals and environment to form a compiled closure. The `.Internal` function `bcClose` does this. Some additional fiddling is needed if the closure is an S4 generic. The need for the `asS4` bit seems a bit odd but it is apparently needed at this point.

78c ⟨cmpfun *function* 78c⟩≡
```
cmpfun <- function(f, options = NULL) {
    type <- typeof(f)
```

```
    if (type == "closure") {
        cntxt <- make.toplevelContext(makeCenv(environment(f)), options)
        ncntxt <- make.functionContext(cntxt, formals(f), body(f))
        b <- genCode(body(f), ncntxt)
        val <- .Internal(bcClose(formals(f), b, environment(f)))
        attrs <- attributes(f)
        if (! is.null(attrs))
            attributes(val) <- attrs
        if (isS4(f)) ## **** should this really be needed??
            val <- asS4(val)
        val
    }
    else if (typeof(f) == "builtin" || type == "special")
        f
    else stop("cannot compile a non-function")
}
```

For use in compiling packages and in JIT compilation it is useful to have a variant that returns the uncompiled function if there is an error during compilation.

79a ⟨`tryCmpfun` *function* 79a⟩≡

```
tryCmpfun <- function(f)
    tryCatch(cmpfun(f), error = function(e) f)
```

15.2 Compiling and loading files

A file can be compiled with `cmpfile` and loaded with `loadcmp`. `cmpfile` reads in the expressions, compiles them, and serializes the list of compiled expressions by calling the `.Internal` function `save.to.file`.

79b ⟨`cmpfile` *function* 79b⟩≡

```
cmpfile <- function(infile, outfile, ascii = FALSE, env = .GlobalEnv,
                    verbose = FALSE, options = NULL) {
    if (! is.environment(env) || ! identical(env, topenv(env)))
        stop("'env' must be a top level environment")
    ⟨create outfile if argument is missing 80a⟩
    ⟨check that infile and outfile are not the same 80b⟩
    forms <- parse(infile)
    nforms <- length(forms)
    if (nforms > 0) {
        expr.needed <- 1000
        expr.old <- options()$expressions
        if (expr.old < expr.needed) {
            options(expressions = expr.needed)
            on.exit(options(expressions = expr.old))
        }
        cforms <- vector("list", nforms)
        cenv <- makeCenv(env)
        cntxt <- make.toplevelContext(cenv, options)
        cntxt$env <- addCenvVars(cenv, findLocalsList(forms, cntxt))
```

```
        for (i in 1:nforms) {
            e <- forms[[i]]
            if (verbose) {
                if (typeof(e) == "language" && e[[1]] == "<-" &&
                    typeof(e[[3]]) == "language" && e[[3]][[1]] == "function")
                    cat(paste0("compiling function \"", e[[2]], "\"\n"))
                else
                    cat(paste("compiling expression", deparse(e, 20)[1],
                              "...\n"))
            }
            cforms[[i]] <- genCode(e, cntxt)
        }
        cat(gettextf("saving to file \"%s\" ... ", outfile))
        .Internal(save.to.file(cforms, outfile, ascii))
        cat(gettext("done"), "\n", sep = "")
    }
    else warning("empty input file; no output written");
    invisible(NULL)
}
```

The default output file name is the base name of the input file with a `.Rc` extension.

80a ⟨*create* `outfile` *if argument is missing* 80a⟩≡

```
if (missing(outfile)) {
    basename <- sub("\\.[a-zA-Z0-9]$", "", infile)
    outfile <- paste0(basename, ".Rc")
}
```

As a precaution it is useful to check that `infile` and `outfile` are not the same and signal an error if they are.

80b ⟨*check that* `infile` *and* `outfile` *are not the same* 80b⟩≡

```
if (infile == outfile)
    stop("input and output file names are the same")
```

The `loadcmp` reads in the serialized list of expressions using the `.Internal` function `load.from.file`. The compiled expressions are then evaluated in the global environment.

80c ⟨`loadcmp` *function* 80c⟩≡

```
loadcmp <- function (file, envir = .GlobalEnv, chdir = FALSE) {
    if (!(is.character(file) && file.exists(file)))
        stop(gettextf("file '%s' does not exist", file), domain = NA)
    exprs <- .Internal(load.from.file(file))
    if (length(exprs) == 0)
        return(invisible())
    if (chdir && (path <- dirname(file)) != ".") {
        owd <- getwd()
        on.exit(setwd(owd), add = TRUE)
        setwd(path)
    }
    for (i in exprs) {
        yy <- eval(i, envir)
```

```
    }
    invisible()
}
```

loadcmp is the analog to source for compiled files.

Two additional functions that are currently not exported or used are cmpframe and cmplib. They should probably be removed.

81a ⟨cmpframe *function* 81a⟩≡

```
cmpframe <- function(inpos, file) {
    expr.needed <- 1000
    expr.old <- options()$expressions
    if (expr.old < expr.needed)
       options(expressions = expr.needed)
    on.exit(options(expressions = expr.old))

    attach(NULL, name="<compiled>")
    inpos <- inpos + 1
    outpos <- 2
    on.exit(detach(pos=outpos), add=TRUE)

    for (f in ls(pos = inpos, all.names = TRUE)) {
        def <- get(f, pos = inpos)
        if (typeof(def) == "closure") {
                cat(gettextf("compiling '%s'", f), "\n", sep = "")
                fc <- cmpfun(def)
                assign(f, fc, pos=outpos)
        }
    }
    cat(gettextf("saving to file \"%s\" ... ", file))
    save(list = ls(pos = outpos, all.names = TRUE), file = file)
    cat(gettext("done"), "\n", sep = "")
}
```

81b ⟨cmplib *function* 81b⟩≡

```
cmplib <- function(package, file) {
    package <- as.character(substitute(package))
    pkgname <- paste("package", package, sep = ":")
    pos <- match(pkgname, search());
    if (missing(file))
        file <- paste0(package,".Rc")
    if (is.na(pos)) {
        library(package, character.only = TRUE)
        pos <- match(pkgname, search());
        on.exit(detach(pos=match(pkgname, search())))
    }
    cmpframe(pos, file)
}
```

15.3 Enabling implicit compilation

82a ⟨enableJIT *function* 82a⟩≡
```
enableJIT <- function(level)
    .Internal(enableJIT(level))
```

82b ⟨compilePKGS *function* 82b⟩≡
```
compilePKGS <- function(enable)
    .Internal(compilePKGS(enable))
```

15.4 Setting compiler options

The `setCompilerOptions` function provides a means for users to adjust the default compiler option values. This interface is experimental and may change.

82c ⟨setCompilerOptions *function* 82c⟩≡
```
setCompilerOptions <- function(...) {
    options <- list(...)
    nm <- names(options)
    for (n in nm)
        if (! exists(n, compilerOptions))
            stop(gettextf("'%s' is not a valid compiler option", n),
                 domain = NA)
    old <- list()
    for (n in nm) {
        op <- options[[n]]
        switch(n,
               optimize = {
                   op <- as.integer(op)
                   if (length(op) == 1 && 0 <= op && op <= 3) {
                       old <- c(old, list(optimize =
                                          compilerOptions$optimize))
                       compilerOptions$optimize <- op
                   }
               },
               suppressAll = {
                   if (identical(op, TRUE) || identical(op, FALSE)) {
                       old <- c(old, list(suppressAll =
                                          compilerOptions$suppressAll))
                       compilerOptions$suppressAll <- op
                   }
               },
               suppressUndefined = {
                   if (identical(op, TRUE) || identical(op, FALSE) ||
                       is.character(op)) {
                       old <- c(old, list(suppressUndefined =
                                          compilerOptions$suppressUndefined))
                       compilerOptions$suppressUndefined <- op
                   }
```

```
            })
        }
        invisible(old)
    }
```

For now, a `.onLoad` function is used to allow all warning to be suppressed. This is probably useful for building packages, since the way lazy loading is done means variables defined in shared libraries are not available and produce a raft of warnings. The `.onLoad` function also allows undefined variables to be suppressed and the optimization level to be specified using environment variables.

83a ⟨`.onLoad` *function* 83a⟩≡

```
.onLoad <- function(libname, pkgname) {
    if (Sys.getenv("R_COMPILER_SUPPRESS_ALL") != "")
        setCompilerOptions(suppressAll = TRUE)
    if (Sys.getenv("R_COMPILER_SUPPRESS_UNDEFINED") != "")
        setCompilerOptions(suppressUndefined = TRUE)
    if (Sys.getenv("R_COMPILER_OPTIMIZE") != "")
        tryCatch({
            lev <- as.integer(Sys.getenv("R_COMPILER_OPTIMIZE"))
            if (0 <= lev && lev <= 3)
                setCompilerOptions(optimize = lev)
        }, error = function(e) e, warning = function(w) w)
}
```

15.5 Disassembler

A minimal disassembler is provided by `disassemble`. This is primarily useful for debugging the compiler. A more readable output representation might be nice to have. It would also probably make sense to give the result a class and write a print method.

83b ⟨`disassemble` *function* 83b⟩≡

```
disassemble <- function(code) {
    .CodeSym <- as.name(".Code")
    disasm.const<-function(x)
        if (typeof(x)=="list" && length(x) > 0 && identical(x[[1]], .CodeSym))
            disasm(x) else x
    disasm <-function(code) {
        code[[2]]<-bcDecode(code[[2]])
        code[[3]]<-lapply(code[[3]], disasm.const)
        code
    }
    if (typeof(code)=="closure") {
        code <- .Internal(bodyCode(code))
        if (typeof(code) != "bytecode")
            stop("function is not compiled")
    }
    dput(disasm(.Internal(disassemble(code))))
}
```

The `.Internal` function `disassemble` extracts the numeric code vector and constant pool. The function `bcDecode` uses the `Opcodes.names` array to translate the numeric opcodes into symbolic ones. At this point not enough information is available in a reasonable place to also convert labels back to symbolic form.

84 ⟨`bcDecode` *function* 84⟩≡
```
bcDecode <- function(code) {
    n <- length(code)
    ncode <- vector("list", n)
    ncode[[1]] <- code[1] # version number
    i <- 2
    while (i <= n) {
        name<-Opcodes.names[code[i]+1]
        argc<-Opcodes.argc[[code[i]+1]]
        ncode[[i]] <- as.name(name)
        i<-i+1
        if (argc > 0)
            for (j in 1:argc) {
                ncode[[i]]<-code[i]
                i<-i+1
            }
    }
    ncode
}
```

16 Discussion and future directions

Despite its long gestation period this compiler should be viewed as a first pass at creating a byte code compiler for R. The compiler itself is very simple in design as a single pass compiler with no separate optimization phases. Similarly the virtual machine uses a very simple stack design. While the compiler already achieves some useful performance improvements on loop-intensive code, more can be achieved with more sophisticated approaches. This will be explored in future work.

A major objective of this first version was to reproduce R's interpreted semantics with as few departures as possible while at the same time optimizing a number of aspect of the execution process. The inlining rules controlled by an optimization level setting seem to provide a good way of doing this, and the default optimization setting seems to be reasonably effective. Mechanisms for adjusting the default settings via declarations will be explored and added in the near future.

Future versions of the compiler and the engine will explore a number of alternative designs. Switching to a register-based virtual machine will be explored fairly soon. Preliminary experiments suggest that this can provide significant improvements in the case of tight loops by allowing allocation of intermediate results to be avoided in many cases. It may be possible at least initially to keep the current compiler ant just translate the stack-based machine code to a register-based code.

Another direction that will be explored is whether sequences of arithmetic and other numerical operations can be fused and possibly vectorized. Again preliminary experiments are promising, but more exploration is needed.

Other improvements to be examined may affect interpreted code as much as compiled code. These include more efficient environment representations and more efficient calling conventions.

A General utility functions

This appendix provides a few general utility functions.

The utility function `pasteExpr` is used in the error messages.

85a ⟨`pasteExpr` *function* 85a⟩≡
```
pasteExpr <- function(e, prefix = "\n    ") {
    de <- deparse(e)
    if (length(de) == 1) sQuote(de)
    else paste(prefix, deparse(e), collapse="")
}
```

The function `dots.or.missing` checks the argument list for any missing or `...` arguments:

85b ⟨`dots.or.missing` *function* 85b⟩≡
```
dots.or.missing <- function(args) {
    for (i in 1:length(args)) {
        a <-args[[i]]
        if (missing(a)) return(TRUE) #**** better test?
        if (typeof(a) == "symbol" && a == "...") return(TRUE)
    }
    return(FALSE)
}
```

The function `any.dots` is defined as

85c ⟨`any.dots` *function* 85c⟩≡
```
any.dots <- function(args) {
    for (i in 1:length(args)) {
        a <-args[[i]]
        if (! missing(a) && typeof(a) == "symbol" && a == "...")
            return(TRUE)
    }
    return(FALSE)
}
```

The utility function `is.ddsym` is used to recognize symbols of the form `..1`, `..2`, and so on.

85d ⟨`is.ddsym` *function* 85d⟩≡
```
is.ddsym <- function(name) {
    (is.symbol(name) || is.character(name)) &&
    length(grep("^\\.\\.[0-9]+$", as.character(name))) != 0
}
```

`missingArgs` takes an argument list for a call a logical vector indicating for each argument whether it is empty (missing) or not.

85e ⟨`missingArgs` *function* 85e⟩≡
```
missingArgs <- function(args) {
    val <- logical(length(args))
```

```
    for (i in seq_along(args)) {
        a <- args[[i]]
        if (missing(a))
            val[i] <- TRUE
        else
            val[i] <- FALSE
    }
    val
}
```

B Environment utilities

This appendix presents some utilities for computations on environments.

The function `frameTypes` takes an environment argument and returns a character vector with elements for each frame in the environment classifying the frame as local, namespace, or global. The environment is assumed to be a standard evaluation environment that contains `.GlobalEnv` as one of its parents. It does this by computing the number of local, namespace, and global frames and then generating the result using `rep`.

86a ⟨`frameTypes` *function* 86a⟩≡

```
frameTypes <- function(env) {
    top <- topenv(env)
    empty <- emptyenv()
    ⟨find the number nl of local frames 86b⟩
    ⟨find the number nn of namespace frames 86c⟩
    ⟨find the number ng of global frames 87a⟩
    rep(c("local", "namespace", "global"), c(nl, nn, ng))
}
```

The number of local frames is computes by marching down the parent frames with `parent.env` until the top level environment is reached.

86b ⟨*find the number* `nl` *of local frames* 86b⟩≡

```
nl <- 0
while (! identical(env, top)) {
    env <- parent.env(env)
    nl <- nl + 1
    if (identical(env, empty))
        stop("not a proper evaluation environment")
}
```

The number of namespace frames is computed by continuing down the parent frames until `.GlobalEnv` is reached.

86c ⟨*find the number* `nn` *of namespace frames* 86c⟩≡

```
nn <- 0
if (isNamespace(env)) {
    while (! identical(env, .GlobalEnv)) {
        env <- parent.env(env)
        nn <- nn + 1
```

```
        if (identical(env, empty))
            stop("not a proper evaluation environment")
    }
}
```

Finally the number of global frames is computed by continuing until the empty environment is reached. An alternative would be to compute the length of the result returned by `search`

87a ⟨*find the number* `ng` *of global frames* 87a⟩≡

```
ng <- 0
while (! identical(env, empty)) {
    env <- parent.env(env)
    ng <- ng + 1
}
```

The function `findHomeNS` takes a variable name and a namespace frame, or a namespace imports frame, and returns the namespace frame in which the variable was originally defined, if any. The code assumes that renaming has not been used (it may no longer be supported in the namespace implementation in any case). Just in case, an attempt is made to check for renaming. The result returned is the namaspace frame for the namespace in which the variable was defined or `NULL` if the variable was not defined in the specified namespace or one of its imports, or if the home namespace cannot be determined.

87b ⟨`findHomeNS` *function* 87b⟩≡

```
## Given a symbol name and a namespace environment (or a namespace
## imports environment) find the namespace in which the symbol's value
## was originally defined. Returns NULL if the symbol is not found via
## the namespace.
findHomeNS <- function(sym, ns) {
    ⟨if ns is an imports frame find the corresponding namespace 87c⟩
    if (exists(sym, ns, inherits = FALSE))
        ns
    else if (exists(".__NAMESPACE__.", ns, inherits = FALSE)) {
        ⟨search the imports for sym 88a⟩
        NULL
    }
    else NULL
}
```

If the `ns` argument is not a namespace frame it should be the imports frame of a namespace. Such an imports frame should have a `name` attribute or the form `"imports:foo"` it it is associated with namespace `"foo"`. This is used to find the namespace frame that owns the imports frame in this case, and this frames is then assigned to `ns`.

87c ⟨*if* `ns` *is an imports frame find the corresponding namespace* 87c⟩≡

```
if (! isNamespace(ns)) {
    ## As a convenience this allows for 'ns' to be the imports fame
    ## of a namespace. It appears that these now have a 'name'
    ## attribute of the form 'imports:foo' if 'foo' is the
    ## namespace.
    name <- attr(ns, "name")
```

```
    if (is.null(name))
        stop("'ns' must be a namespace or a namespace imports environment")
    ns <- getNamespace(sub("imports:", "", attr(ns, "name")))
}
```

The imports are searched in reverse order since in the case of name conflicts the last one imported will take precedence. Full imports via an `import` directive have to be handled differently than selective imports created with `importFrom` directives.

88a ⟨*search the imports for* `sym` 88a⟩≡

```
imports <- get(".__NAMESPACE__.", ns)$imports
for (i in rev(seq_along(imports))) {
    iname <- names(imports)[i]
    ins <- getNamespace(iname)
    if (identical(imports[[i]], TRUE)) {
        ⟨search in a full import 88b⟩
    }
    else {
        ⟨search in a selective import 88c⟩
    }
}
```

If an entry in the `imports` specification for the import source namespace `ins` has value `TRUE`, then all exports of the `ins` have been imported. If `sym` is in the exports then the result of a recursive call to `findHomeNS` is returned.

88b ⟨*search in a full import* 88b⟩≡

```
if (identical(ins, .BaseNamespaceEnv))
    exports <- .BaseNamespaceEnv
else
    exports <- get(".__NAMESPACE__.", ins)$exports
if (exists(sym, exports, inherits = FALSE))
    return(findHomeNS(sym, ins))
```

For selective imports the `imports` entry is a named character vector mapping export name to import name. In the absence of renaming the names should match the values; if this is not the case `NULL` is returned. Otherwise, a match results again in returning a recursive call to `findHomeNS`.

88c ⟨*search in a selective import* 88c⟩≡

```
exports <- imports[[i]]
pos <- match(sym, names(exports), 0)
if (pos) {
    ## If renaming has been used things get too
    ## confusing so return NULL. (It is not clear if
    ## renaming this is still supported by the
    ## namespace code.)
    if (sym == exports[pos])
        return(findHomeNS(sym, ins))
    else
        return(NULL)
}
```

Given a package package frame from the global environment the function `packFrameName` returns the associated package name, which is computed from the `name` attribute.

89a ⟨`packFrameName` *function* 89a⟩≡
```
packFrameName <- function(frame) {
    fname <- attr(frame, "name")
    if (is.character(fname))
        sub("package:", "", fname)
    else if (identical(frame , baseenv()))
        "base"
    else ""
}
```

For a namespace frame the function `nsName` retrieves the namespace name from the namespace information structure.

89b ⟨`nsName` *function* 89b⟩≡
```
nsName <- function(ns) {
    if (identical(ns, .BaseNamespaceEnv))
        "base"
    else {
        name <- ns$.__NAMESPACE__.$spec["name"]
        if (is.character(name))
            as.character(name) ## strip off names
        else ""
    }
}
```

C Experimental utilities

This section presents two experimental utililities that, for now, are not exported. The first is a simple byte code profiler. This requires that the file `eval.c` be compiled with `BC_PROFILING` enabled, which on `gcc`-compatible compilers will disable threaded code. The byte code profiler uses the profile timer to record the active byte code instruction at interrupt time. The function `bcprof` runs the profiler while evaluating its argument expression and returns a summary of the counts.

89c ⟨`bcprof` *function* 89c⟩≡
```
bcprof <- function(expr) {
    .Internal(bcprofstart())
    expr
    .Internal(bcprofstop())
    val <- structure(.Internal(bcprofcounts()),
                     names = compiler:::Opcodes.names)
    hits <- sort(val[val > 0], decreasing = TRUE)
    pct <- round(100 * hits / sum(hits), 1)
    data.frame(hits = hits, pct = pct)
}
```

The second utility is a simple interface to the code building mechanism that may help with experimenting with code optimizations.

90a ⟨asm *function* 90a⟩≡

```
asm <- function(e, gen, env = .GlobalEnv, options = NULL) {
    cenv <- makeCenv(env)
    cntxt <- make.toplevelContext(cenv, options)
    cntxt$env <- addCenvVars(cenv, findLocals(e, cntxt))
    genCode(e, cntxt, gen = gen)
}
```

D Opcode constants

D.1 Symbolic opcode names

90b ⟨*opcode definitions* 90b⟩≡

```
BCMISMATCH.OP <- 0
RETURN.OP <- 1
GOTO.OP <- 2
BRIFNOT.OP <- 3
POP.OP <- 4
DUP.OP <- 5
PRINTVALUE.OP <- 6
STARTLOOPCNTXT.OP <- 7
ENDLOOPCNTXT.OP <- 8
DOLOOPNEXT.OP <- 9
DOLOOPBREAK.OP <- 10
STARTFOR.OP <- 11
STEPFOR.OP <- 12
ENDFOR.OP <- 13
SETLOOPVAL.OP <- 14
INVISIBLE.OP <- 15
LDCONST.OP <- 16
LDNULL.OP <- 17
LDTRUE.OP <- 18
LDFALSE.OP <- 19
GETVAR.OP <- 20
DDVAL.OP <- 21
SETVAR.OP <- 22
GETFUN.OP <- 23
GETGLOBFUN.OP <- 24
GETSYMFUN.OP <- 25
GETBUILTIN.OP <- 26
GETINTLBUILTIN.OP <- 27
CHECKFUN.OP <- 28
MAKEPROM.OP <- 29
DOMISSING.OP <- 30
SETTAG.OP <- 31
```

```
DODOTS.OP <- 32
PUSHARG.OP <- 33
PUSHCONSTARG.OP <- 34
PUSHNULLARG.OP <- 35
PUSHTRUEARG.OP <- 36
PUSHFALSEARG.OP <- 37
CALL.OP <- 38
CALLBUILTIN.OP <- 39
CALLSPECIAL.OP <- 40
MAKECLOSURE.OP <- 41
UMINUS.OP <- 42
UPLUS.OP <- 43
ADD.OP <- 44
SUB.OP <- 45
MUL.OP <- 46
DIV.OP <- 47
EXPT.OP <- 48
SQRT.OP <- 49
EXP.OP <- 50
EQ.OP <- 51
NE.OP <- 52
LT.OP <- 53
LE.OP <- 54
GE.OP <- 55
GT.OP <- 56
AND.OP <- 57
OR.OP <- 58
NOT.OP <- 59
DOTSERR.OP <- 60
STARTASSIGN.OP <- 61
ENDASSIGN.OP <- 62
STARTSUBSET.OP <- 63
DFLTSUBSET.OP <- 64
STARTSUBASSIGN.OP <- 65
DFLTSUBASSIGN.OP <- 66
STARTC.OP <- 67
DFLTC.OP <- 68
STARTSUBSET2.OP <- 69
DFLTSUBSET2.OP <- 70
STARTSUBASSIGN2.OP <- 71
DFLTSUBASSIGN2.OP <- 72
DOLLAR.OP <- 73
DOLLARGETS.OP <- 74
ISNULL.OP <- 75
ISLOGICAL.OP <- 76
ISINTEGER.OP <- 77
ISDOUBLE.OP <- 78
ISCOMPLEX.OP <- 79
```

```
ISCHARACTER.OP <- 80
ISSYMBOL.OP <- 81
ISOBJECT.OP <- 82
ISNUMERIC.OP <- 83
VECSUBSET.OP <- 84
MATSUBSET.OP <- 85
VECSUBASSIGN.OP <- 86
MATSUBASSIGN.OP <- 87
AND1ST.OP <- 88
AND2ND.OP <- 89
OR1ST.OP <- 90
OR2ND.OP <- 91
GETVAR_MISSOK.OP <- 92
DDVAL_MISSOK.OP <- 93
VISIBLE.OP <- 94
SETVAR2.OP <- 95
STARTASSIGN2.OP <- 96
ENDASSIGN2.OP <- 97
SETTER_CALL.OP <- 98
GETTER_CALL.OP <- 99
SWAP.OP <- 100
DUP2ND.OP <- 101
SWITCH.OP <- 102
RETURNJMP.OP <- 103
STARTSUBSET_N.OP <- 104
STARTSUBASSIGN_N.OP <- 105
VECSUBSET2.OP <- 106
MATSUBSET2.OP <- 107
VECSUBASSIGN2.OP <- 108
MATSUBASSIGN2.OP <- 109
STARTSUBSET2_N.OP <- 110
STARTSUBASSIGN2_N.OP <- 111
SUBSET_N.OP <- 112
SUBSET2_N.OP <- 113
SUBASSIGN_N.OP <- 114
SUBASSIGN2_N.OP <-115
LOG.OP <- 116
LOGBASE.OP <- 117
MATH1.OP <- 118
DOTCALL.OP <- 119
COLON.OP <- 120
SEQALONG.OP <- 121
SEQLEN.OP <- 122
```

D.2 Instruction argument counts and names

92 ⟨*opcode argument counts* 92⟩≡

```
Opcodes.argc <- list(
```

```
BCMISMATCH.OP = 0,
RETURN.OP = 0,
GOTO.OP = 1,
BRIFNOT.OP = 2,
POP.OP = 0,
DUP.OP = 0,
PRINTVALUE.OP = 0,
STARTLOOPCNTXT.OP = 1,
ENDLOOPCNTXT.OP = 0,
DOLOOPNEXT.OP = 0,
DOLOOPBREAK.OP = 0,
STARTFOR.OP = 3,
STEPFOR.OP = 1,
ENDFOR.OP = 0,
SETLOOPVAL.OP = 0,
INVISIBLE.OP = 0,
LDCONST.OP = 1,
LDNULL.OP = 0,
LDTRUE.OP = 0,
LDFALSE.OP = 0,
GETVAR.OP = 1,
DDVAL.OP = 1,
SETVAR.OP = 1,
GETFUN.OP = 1,
GETGLOBFUN.OP = 1,
GETSYMFUN.OP = 1,
GETBUILTIN.OP = 1,
GETINTLBUILTIN.OP = 1,
CHECKFUN.OP = 0,
MAKEPROM.OP = 1,
DOMISSING.OP = 0,
SETTAG.OP = 1,
DODOTS.OP = 0,
PUSHARG.OP = 0,
PUSHCONSTARG.OP = 1,
PUSHNULLARG.OP = 0,
PUSHTRUEARG.OP = 0,
PUSHFALSEARG.OP = 0,
CALL.OP = 1,
CALLBUILTIN.OP = 1,
CALLSPECIAL.OP = 1,
MAKECLOSURE.OP = 1,
UMINUS.OP = 1,
UPLUS.OP = 1,
ADD.OP = 1,
SUB.OP = 1,
MUL.OP = 1,
DIV.OP = 1,
```

```
EXPT.OP = 1,
SQRT.OP = 1,
EXP.OP = 1,
EQ.OP = 1,
NE.OP = 1,
LT.OP = 1,
LE.OP = 1,
GE.OP = 1,
GT.OP = 1,
AND.OP = 1,
OR.OP = 1,
NOT.OP = 1,
DOTSERR.OP = 0,
STARTASSIGN.OP = 1,
ENDASSIGN.OP = 1,
STARTSUBSET.OP = 2,
DFLTSUBSET.OP = 0,
STARTSUBASSIGN.OP = 2,
DFLTSUBASSIGN.OP = 0,
STARTC.OP = 2,
DFLTC.OP = 0,
STARTSUBSET2.OP = 2,
DFLTSUBSET2.OP = 0,
STARTSUBASSIGN2.OP = 2,
DFLTSUBASSIGN2.OP = 0,
DOLLAR.OP = 2,
DOLLARGETS.OP = 2,
ISNULL.OP = 0,
ISLOGICAL.OP = 0,
ISINTEGER.OP = 0,
ISDOUBLE.OP = 0,
ISCOMPLEX.OP = 0,
ISCHARACTER.OP = 0,
ISSYMBOL.OP = 0,
ISOBJECT.OP = 0,
ISNUMERIC.OP = 0,
VECSUBSET.OP = 1,
MATSUBSET.OP = 1,
VECSUBASSIGN.OP = 1,
MATSUBASSIGN.OP = 1,
AND1ST.OP = 2,
AND2ND.OP = 1,
OR1ST.OP = 2,
OR2ND.OP = 1,
GETVAR_MISSOK.OP = 1,
DDVAL_MISSOK.OP = 1,
VISIBLE.OP = 0,
SETVAR2.OP = 1,
```

```
  STARTASSIGN2.OP = 1,
  ENDASSIGN2.OP = 1,
  SETTER_CALL.OP = 2,
  GETTER_CALL.OP = 1,
  SWAP.OP = 0,
  DUP2ND.OP = 0,
  SWITCH.OP = 4,
  RETURNJMP.OP = 0,
  STARTSUBSET_N.OP = 2,
  STARTSUBASSIGN_N.OP = 2,
  VECSUBSET2.OP = 1,
  MATSUBSET2.OP = 1,
  VECSUBASSIGN2.OP = 1,
  MATSUBASSIGN2.OP = 1,
  STARTSUBSET2_N.OP = 2,
  STARTSUBASSIGN2_N.OP = 2,
  SUBSET_N.OP = 2,
  SUBSET2_N.OP = 2,
  SUBASSIGN_N.OP = 2,
  SUBASSIGN2_N.OP = 2,
  LOG.OP = 1,
  LOGBASE.OP = 1,
  MATH1.OP = 2,
  DOTCALL.OP = 2,
  COLON.OP = 1,
  SEQALONG.OP = 1,
  SEQLEN.OP = 1
  )
```

95a ⟨*opcode names* 95a⟩≡

```
  Opcodes.names <- names(Opcodes.argc)
```

E Implementation file

95b ⟨*cmp.R* 95b⟩≡

```
  # Automatically generated from ../noweb/compiler.nw.
  #
  # File src/library/compiler/R/cmp.R
  # Part of the R package, http://www.R-project.org
  # Copyright (C) 2001-2014 Luke Tierney
  #
  # 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 2 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.
#
#  A copy of the GNU General Public License is available at
#  http://www.r-project.org/Licenses/

##
## Compiler options
##

⟨compiler options data base 18b⟩

⟨getCompilerOption function 18c⟩


##
## General Utilities
##

⟨pasteExpr function 85a⟩

⟨dots.or.missing function 85b⟩

⟨any.dots function 85c⟩

⟨is.ddsym function 85d⟩


⟨missingArgs function 85e⟩


##
## Environment utilities
##

⟨frameTypes function 86a⟩

⟨findHomeNS function 87b⟩

⟨packFrameName function 89a⟩

⟨nsName function 89b⟩


##
## Finding possible local variables
##
```

```
⟨getAssignedVar function 26a⟩

⟨findLocals1 function 27b⟩

⟨findLocalsList1 function 27c⟩

⟨findLocals function 29a⟩

⟨findLocalsList function 28b⟩


##
## Compilation environment implementation
##

⟨makeCenv function 22a⟩

⟨addCenvVars function 22b⟩

⟨addCenvFrame function 22c⟩

⟨findCenvVar function 23a⟩

⟨isBaseVar function 29b⟩

⟨funEnv function 29c⟩

⟨findLocVar function 24c⟩

⟨findFunDef function 24d⟩

⟨findVar function 24b⟩


##
## Constant folding
##

⟨maxConstSize and constModes definitions 76c⟩

⟨constNames definition 77b⟩

⟨checkConst function 76b⟩

⟨constantFoldSym function 77a⟩

⟨getFoldFun function 78b⟩
```

```
⟨constantFoldCall function 77c⟩

⟨constantFold function 76a⟩

⟨foldFuns definition 78a⟩

⟨languageFuns definition 30a⟩


##
## Opcode constants
##

⟨opcode argument counts 92⟩

⟨opcode names 95a⟩

⟨opcode definitions 90b⟩


##
## Code buffer implementation
##

⟨make.codeBuf function 13⟩

⟨codeBufCode function 15c⟩

⟨genCode function 4a⟩


##
## Compiler contexts
##

⟨make.toplevelContext function 16a⟩

⟨make.callContext function 16b⟩

⟨make.promiseContext function 18a⟩

⟨make.functionContext function 17b⟩

⟨make.nonTailCallContext function 16c⟩

⟨make.argContext function 17d⟩
```

```
⟨make.noValueContext function 17a⟩

⟨make.loopContext function 17c⟩


##
## Compiler top level
##

⟨cmp function 5b⟩

⟨cmpConst function 6⟩

⟨cmpSym function 7a⟩

⟨cmpCall function 9b⟩

⟨cmpCallSymFun function 8b⟩

⟨cmpCallExprFun function 9a⟩

⟨cmpCallArgs function 11a⟩

⟨cmpConstArg 12b⟩

⟨checkCall function 10a⟩

## **** need to handle ... and ..n arguments specially
## **** separate call opcode for calls with named args?
## **** for (a in e[[-1]]) ... goes into infinite loop

⟨cmpTag function 12c⟩


##
## Inlining mechanism
##

⟨inline handler implementation 31c⟩

## tryInline implements the rule permitting inlining as they stand now:
## Inlining is controlled by the optimize compiler option, with possible
## values 0, 1, 2, 3.

⟨getInlineInfo function 31a⟩

⟨tryInline function 30b⟩
```

```
##
## Inline handlers for some SPECIAL functions
##

⟨inlining handler for function 36⟩

⟨inlining handler for left brace function 35c⟩

⟨inlining handler for if 40d⟩

⟨inlining handler for && 41a⟩

⟨inlining handler for || 41b⟩


##
## Inline handlers for assignment expressions
##

⟨setter inlining mechanism 70c⟩

⟨getter inlining mechanism 74a⟩

⟨cmpAssign function 63a⟩

⟨flattenPlace function 68b⟩

⟨cmpGetterCall function 73a⟩

⟨checkAssign function 63c⟩

⟨cmpSymbolAssign function 64b⟩

⟨cmpComplexAssign function 67a⟩

⟨cmpSetterCall function 69b⟩

⟨getAssignFun function 70b⟩

⟨cmpSetterDispatch function 72a⟩

⟨inlining handlers for <-, =, and <<- 63b⟩

⟨setter inline handler for $<- 71⟩

⟨setter inline handlers for [<- and [[<- 72b⟩
```

```
⟨cmpGetterDispatch function 75a⟩

⟨getter inline handler for $ 74b⟩

⟨getter inline handlers for [ and [[ 75b⟩


##
## Inline handlers for loops
##

⟨inlining handlers for next and break 44a⟩

⟨isLoopStopFun function 48c⟩

⟨isLoopTopFun function 48d⟩

⟨checkSkipLoopCntxtList function 48b⟩

⟨checkSkipLoopCntxt function 48a⟩

⟨inlining handler for repeat loops 42⟩

⟨cmpRepeatBody function 43b⟩

⟨inlining handler for while loops 44b⟩

⟨cmpWhileBody function 45⟩

⟨inlining handler for for loops 46a⟩

⟨cmpForBody function 47a⟩


##
## Inline handlers for one and two argument primitives
##

⟨cmpPrim1 function 49c⟩

⟨cmpPrim2 function 50a⟩

⟨inline handlers for + and - 49a⟩

⟨inline handlers for * and / 49b⟩

⟨inline handlers for ^, exp, and sqrt 50b⟩
```

```
⟨inline handlers for comparison operators 51a⟩

⟨inline handlers for & and | 51b⟩

⟨inline handler for ! 51c⟩


##
## Inline handlers for the left parenthesis function
##

⟨inlining handler for ( 37a⟩


##
## Inline handlers for general BUILTIN and SPECIAL functions
##

⟨cmpBuiltin function 33a⟩

⟨cmpBuiltinArgs function 33b⟩

⟨cmpSpecial function 35a⟩

⟨inlining handler for .Internal 37b⟩


##
## Inline handlers for subsetting and related operators
##

⟨cmpDispatch function 52b⟩

⟨inlining handlers for some dispatching SPECIAL functions 52a⟩

⟨inlining handler for $ 53⟩


##
## Inline handler for local() and return() functions
##

⟨inlining handler for local function 38a⟩

⟨inlining handler for return function 38b⟩


##
```

```
## Inline handlers for the family of is.xyz primitives
##

⟨cmpIs function 56c⟩

⟨inlining handlers for is.xyz functions 57a⟩


##
## Default inline handlers for BUILTIN and SPECIAL functions
##

⟨install default inlining handlers 32⟩


##
## Inline handlers for some .Internal functions
##

⟨simpleFormals function 56a⟩

⟨simpleArgs function 56b⟩

⟨is.simpleInternal function 55c⟩

⟨inlineSimpleInternalCall function 54a⟩

⟨cmpSimpleInternal function 54b⟩

⟨inline safe simple .Internal functions from base 54c⟩

⟨inline safe simple .Internal functions from stats 55a⟩


##
## Inline handler for switch
##

⟨findActionIndex function 61b⟩

⟨inline handler for switch 59a⟩


##
## Inline handlers to control warnings
##

⟨cmpMultiColon function 57b⟩
```

```
⟨inlining handlers for :: and ::: 58a⟩

⟨setter inlining handler for @<- 72c⟩

⟨inlining handler for with 58b⟩

⟨inlining handler for require 58c⟩


##
## Compiler warnings
##

⟨suppressAll function 18d⟩

⟨suppressUndef function 18e⟩

⟨notifyLocalFun function 19c⟩

⟨notifyUndefFun function 19a⟩

⟨notifyUndefVar function 19b⟩

⟨notifyNoSuperAssignVar function 20e⟩

⟨notifyWrongArgCount function 20b⟩

⟨notifyWrongDotsUse function 20a⟩

⟨notifyWrongBreakNext function 20d⟩

⟨notifyBadCall function 20c⟩

⟨notifyBadAssignFun function 20f⟩

⟨notifyMultipleSwitchDefaults function 21⟩


##
## Compiler interface
##

⟨compile function 3⟩

⟨cmpfun function 78c⟩

⟨tryCmpfun function 79a⟩
```

```
⟨cmpframe function 81a⟩

⟨cmplib function 81b⟩

⟨cmpfile function 79b⟩

⟨loadcmp function 80c⟩

⟨enableJIT function 82a⟩

⟨compilePKGS function 82b⟩

⟨setCompilerOptions function 82c⟩

⟨.onLoad function 83a⟩


##
## Disassembler
##

⟨bcDecode function 84⟩

⟨disassemble function 83b⟩


##
## Experimental Utilities
##

⟨bcprof function 89c⟩

⟨asm function 90a⟩
```