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

```r
> 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

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

then a compiled version is produced by

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

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

```r
> x <- as.double(1 : 1000000)
> 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

```bash
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.

```r
⟨compile function⟩3 ≡ (102b)

\[
\text{compile} \leftarrow \text{function}(e, \text{env} = \text{.GlobalEnv}, \text{options} = \text{NULL}) \{ \\
\text{cenv} \leftarrow \text{makeCenv}(\text{env}) \\
\text{cntxt} \leftarrow \text{make.toplevelContext}(\text{cenv}, \text{options}) \\
\text{cntxt}\$\text{env} \leftarrow \text{addCenvVars}(\text{cenv}, \text{findLocals}(e, \text{cntxt})) \\
\text{genCode}(e, \text{cntxt}) \\
\}
\]
```

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

```r
⟨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

```r
⟨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 `constantFold` 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.

\[
\text{if (typeof(e) == "language")}
\]
\[
\text{cmpCall(e, cb, cntxt)}
\]
\[
\text{else if (typeof(e) == "symbol")}
\]
\[
\text{cmpSym(e, cb, cntxt, missingOK)}
\]
\[
\text{else if (typeof(e) == "bytecode")}
\]
\[
\text{ctx$stop(gettext("cannot compile byte code literals in code"), cntxt)}
\]
\[
\text{else if (typeof(e) == "promise")}
\]
\[
\text{ctx$stop(gettext("cannot compile promise literals in code"), cntxt)}
\]
\[
\text{else}
\]
\[
\text{cmpConst(e, cb, cntxt)}
\]

The function \text{cmp} is then defined as

\[
\text{cmp <- function(e, cb, cntxt, missingOK = FALSE) {}
}
\]
\[
\text{ce <- constantFold(e, cntxt)}
\]
\[
\text{if (is.null(ce)) {}}
\]
\[
\text{\langle generate code for expression e \rangle \equiv (5a)}
\]
\[
\text{\langle generate code for expression e \rangle \equiv (5b)}
\]
\[
\text{cmpConst(ce$value, cb, cntxt)}
\]
\[
\text{}}
\]

The call code generator \text{cmpCall} will recursively call \text{cmp}.

2.4 Compiling constant expressions

The constant code generator \text{cmpConst} is the simplest of the three generators. A simplified generator can be defined as

\[
\text{\langle simplified cmpConst function \rangle \equiv}
\]
\[
\text{cmpConst <- function(val, cb, cntxt) {}
}
\]
\[
\text{ci <- cb$putconst(val)}
\]
\[
\text{cb$putcode(LDCONST.OP, ci)}
\]
\[
\text{if (cntxt$tailcall) cb$putcode(RETURN.OP)}
\]
\[
\text{}}
\]

This function enters the constant in the constant pool using the closure \text{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 \texttt{RETURN} instruction is emitted as well.

Certain constant values, such as \texttt{TRUE}, \texttt{FALSE}, and \texttt{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 \texttt{cmpConst} that makes use of instructions for loading these particular values is given by

\begin{verbatim}
3\langle cmpConst function\rangle ≡
\begin{verbatim}
   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)
   }
\end{verbatim}
\end{verbatim}
\end{verbatim}

It might be useful to handle other constants in a similar way, such as \texttt{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 \texttt{NAMED} field to 2, but experience in testing shows that there are several packages in the wild that assume that an expression \texttt{TRUE}, for example, appearing in code will result in a freshly allocated value that can be freely modified in \texttt{.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 \texttt{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 \texttt{RETURN} instruction.

In addition to standard variables there is the ellipsis variable \ldots and the accessors \ldots, and so on that need to be considered. The ellipsis variable can only appear as an argument in function calls, so \texttt{cmp} like the interpreter \texttt{eval} itself, should not encounter it. The interpreter signals an error if it does encounter a \ldots 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 \texttt{x[]}). To support this, \texttt{cmpSym} takes an optional argument for allowing
missing argument values.

\[ \text{cmpSym} \ \text{function} \Rightarrow \]
\[ \text{cmpSym} \leftarrow \text{function}(\text{sym}, \text{cb}, \text{ctxt}, \text{missingOK} = \text{FALSE}) \} \]
\[ \text{if} \ (\text{sym} == \ldots) \} \]
\[ \text{if} \ \text{notifyWrongDotsUse}(\ldots, \text{ctxt}) \]
\[ \text{cb}$\text{putcode}(\text{DOTSERR.OP}) \] 
\[ \text{else} \text{if} \ (\text{is.ddsym}(\text{sym})) \} \]
\[ \langle \text{emit code for ..n variable references} \rangle 
\[ \text{else} \} \]
\[ \langle \text{emit code for standard variable references} \rangle 
\[ \} 

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 \text{cmpSym}.

\[ \langle \text{emit code for ..n variable references} \rangle \]
\[ \text{if} \ \text{findLocVar}(\ldots, \text{ctxt}) \]
\[ \text{notifyWrongDotsUse}(\text{sym}, \text{ctxt}) \]
\[ \text{ci} \leftarrow \text{cb}$\text{putconst}(\text{sym}) \]
\[ \text{if} \ (\text{missingOK}) \]
\[ \text{cb}$\text{putcode}(\text{DDVAL_MISSOK.OP, ci}) \]
\[ \text{else} \]
\[ \text{cb}$\text{putcode}(\text{DDVAL.OP, ci}) \]
\[ \text{if} \ (\text{ctxt}$\text{tailcall}) \text{cb}$\text{putcode}(\text{RETURN.OP}) \]

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.

\[ \langle \text{emit code for standard variable references} \rangle \]
\[ \text{if} \ \text{findVar}(\text{sym}, \text{ctxt}) \]
\[ \text{notifyUndefVar}(\text{sym}, \text{ctxt}) \]
\[ \text{ci} \leftarrow \text{cb}$\text{putconst}(\text{sym}) \]
\[ \text{if} \ (\text{missingOK}) \]
\[ \text{cb}$\text{putcode}(\text{GETVAR_MISSOK.OP, ci}) \]
\[ \text{else} \]
\[ \text{cb}$\text{putcode}(\text{GETVAR.OP, ci}) \]
\[ \text{if} \ (\text{ctxt}$\text{tailcall}) \text{cb}$\text{putcode}(\text{RETURN.OP}) \]

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 \texttt{cmpCall}. A simplified version is defined as

\begin{verbatim}
8a \langle simplified cmpCall function \rangle
\texttt{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)
}}
\end{verbatim}

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

\begin{verbatim}
8b \langle cmpCallSymFun function \rangle
\texttt{cmpCallSymFun <- function(fun, args, call, cb, cntxt) {
  ci <- cb$putconst(fun)
  cb$putcode(GETFUN.OP, ci)
  \langle compile arguments and emit CALL instruction \rangle
}}
\end{verbatim}

The \texttt{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 \texttt{cmpCallArgs}, presented in Section 2.7, and is followed by emitting code to execute the call and, if necessary, return a result.

\begin{verbatim}
8c \langle compile arguments and emit CALL instruction \rangle
\texttt{cmpCallArgs(args, cb, cntxt) ci <- cb$putconst(call)
  cb$putcode(CALL.OP, ci)
  if (cntxt$tailcall) cb$putcode(RETURN.OP)}
\end{verbatim}
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.

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

```
cmpCall <- function(call, cb, cntxt) {
  ctx <- make.callContext(cntxt, call)
  fun <- call[[1]]
  args <- call[-1]
  if (typeof(fun) == "symbol") {
    if (!tryInline(call, cb, cntxt)) {
      cmpCallSymFun(fun, args, call, cb, cntxt)
    }
  } else {
    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

```
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))
}

checkCall is defined as

\[ \langle \text{checkCall function} \rangle \equiv \langle \text{tryCatch} \rangle \]

## **** 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

```r
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 <- getOption("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)
    }
  } 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.

\[ \langle \text{hack for handling break() and next() expressions} \rangle \equiv \langle \text{GOTO} \rangle \]

## **** 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:

```r
⟨cmpCallArgs function⟩≡
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⟩
    ⟨compile ... argument⟩
    ⟨signal an error for promise or bytecode argument⟩
    ⟨compile a general argument⟩
  }
}
```

The missing argument case is handled by

```r
⟨compile missing argument⟩≡
if (missing(a)) { ## better test for missing??
  cb$putcode(DOMISSING.OP)
  cmpTag(a, 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:

```r
⟨compile ... argument⟩≡
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.

```r
⟨signal an error for promise or bytecode argument⟩≡
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 != "") {
```
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.

```r
make.codeBuf <- function(expr) {
  ci <- cb$putconst(as.name(n))
  cb$putcode(SETTAG.OP, ci)
  }
}
```

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.
const = getconst,
putcode = putcode,
putconst = putconst,
makelabel = makelabel,
putlabel = putlabel,
patchlabels = patchlabels)

cb$putconst(expr) ## insert expression as first constant.

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.

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

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 \text{identical}, its existing index is returned. Again a size-doubling strategy
is used for the buffer. \text{.Internal} functions are used both for performance reasons and to prevent
duplication of the constants.

\[
\text{constBuf} \leftarrow \text{vector("list", 1})
\]
\[
\text{constCount} \leftarrow 0
\]
\[
\text{putconst} \leftarrow \text{function(x) \{ 
\text{if (constCount == length(constBuf))} 
\text{constBuf} \leftarrow \text{.Internal(growconst(constBuf))} 
\text{i} \leftarrow \text{.Internal(putconst(constBuf, constCount, x))}
\text{if (i == constCount)} 
\text{constCount} \leftarrow \text{constCount + 1}
\text{i}
\}
\]
\[
\text{getconst} \leftarrow \text{function()} 
\text{.Internal(getconst(constBuf, constCount))}
\]

Labels are used for identifying targets for branching instruction. The label management
interface creates new labels with \text{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.

```r
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 or in lists that have been placed in the instruction stream; this is used for the `SWITCH` instruction.

```r
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`.

```r
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⟩

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⟩

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⟩

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 `if` calls and for compiling the bodies of loops.

\[\text{make.noValueContext function} \equiv\]
\begin{verbatim}
make.noValueContext <- function(cntxt) {
  cntxt$tailcall <- FALSE
  cntxt
}
\end{verbatim}

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

\[\text{make.functionContext function} \equiv\]
\begin{verbatim}
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
}
\end{verbatim}

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

\[\text{make.loopContext function} \equiv\]
\begin{verbatim}
make.loopContext <- function(cntxt, loop.label, end.label) {
  ncntxt <- make.noValueContext(cntxt)
  ncntxt$loop <- list(loop = loop.label, end = end.label, gotoOK = TRUE)
  ncntxt
}
\end{verbatim}

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 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.

\[\text{make.argContext function} \equiv\]
\begin{verbatim}
make.argContext <- function(cntxt) {
  cntxt$tailcall <- FALSE
  if (! is.null(cntxt$loop))
    cntxt$loop$gotoOK <- FALSE
  cntxt
}
\end{verbatim}
make.promiseContext <- function(cntxt) {
cntxt$tailcall <- TRUE
cntxt$needRETURNJMP <- TRUE
if (!is.null(cntxt$loop))
  cntxt$loop$gotoOK <- FALSE
  context
}

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.

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.

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 suppressAll property set to TRUE.

suppressAll <- function(cntxt) identical(cntxt$suppressAll, TRUE)

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

suppressUndefined <- function(name, cntxt) {
  if (identical(cntxt$suppressUndefined, TRUE))
    TRUE
  else {
    suppress <- cntxt$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.

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

```r
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`.

```r
notifyLocalFun <- function(fun, cntxt) {
    if (! suppressAll(cntxt))
        NULL
}
```
Warnings about possible improper use of . . . and . . . n variables are sent by `notifyWrongDotsUse`.

\begin{verbatim}
notifyWrongDotsUse <- function(var, cntxt) {
    if (! suppressAll(cntxt))
        cntxt$warn(paste(var, "may be used in an incorrect context"), cntxt)
}
\end{verbatim}

Wrong argument count issues are signaled by `notifyWrongArgCount`.

\begin{verbatim}
notifyWrongArgCount <- function(fun, cntxt) {
    if (! suppressAll(cntxt))
        cntxt$warn(gettextf("wrong number of arguments to '%s'",
                              as.character(fun)),
                  cntxt)
}
\end{verbatim}

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`.

\begin{verbatim}
notifyBadCall <- function(w, cntxt) {
    if (! suppressAll(cntxt))
        cntxt$warn(w, cntxt)
}
\end{verbatim}

`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.

\begin{verbatim}
notifyWrongBreakNext <- function(fun, cntxt) {
    if (! suppressAll(cntxt)) {
        msg <- paste(fun, "may be used in wrong context: no loop is visible")
        cntxt$warn(msg, cntxt)
    }
}
\end{verbatim}

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

\begin{verbatim}
notifyNoSuperAssignVar <- function(symbol, cntxt) {
    if (! suppressAll(cntxt)) {
        msg <- gettextf("no visible binding for '<<-' assignment to '%s'",
                        as.character(symbol))
        cntxt$warn(msg, cntxt)
    }
}
\end{verbatim}

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.
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.

\[
\begin{align*}
\langle \text{notifyMultipleSwitchDefaults function}\rangle &\equiv \langle \text{notifyNoSwitchcases function}\rangle \\
\text{notifyMultipleSwitchDefaults} \leftarrow \text{function}(\text{ndflt}, \text{cntxt}) &\quad \text{if} \quad (! \text{suppressAll}(\text{cntxt})) \\
\text{notifyNoSwitchcases} \leftarrow \text{function}(\text{cntxt}) &\quad \text{if} \quad (! \text{suppressAll}(\text{cntxt}))
\end{align*}
\]

5 Compilation environments

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

\[
\text{if} \ (x > 0) \ \log(x) \ \text{else} \ 0
\]

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 would 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 described in Section 6.

For the compiler to be able to decide whether it 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 \( \text{ftype} \) classifying each frame as a local, namespace, or global frame.

```
\begin{verbatim}
makeCenv <- function(env) {
  structure(list(extra = list(character(0)),
                   env = env,
                   ftypes = frameTypes(env)),
             class = "compiler_environment")
}
\end{verbatim}
```

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.

```
\begin{verbatim}
addCenvVars <- function(cenv, vars) {
  cenv$extra[[1]] <- union(cenv$extra[[1]], vars)
  cenv
}
\end{verbatim}
```

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 \( \text{vars} \) argument is provided so this can be done without an additional call to \( \text{addCenvVars} \).

```
\begin{verbatim}
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.

```r
findCenvVar <- function(var, cenv) {
  if (typeof(var) == "symbol")
    var <- as.character(var)
  extra <- cenv$extra
  env <- cenv$env
  frame <- NULL
  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 (is.null(frame)) {
    empty <- emptyenv()
    while (! identical(env, empty)) {
      i <- i + 1
      if (exists(var, env, inherits = FALSE)) {
        frame <- env
        break
      }
    }
  }
  return.frame(ftype = "local", value = NULL, frame = frame, index = 1)
}
```

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.

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.

```r
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)
}
```
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.

```r
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.

```r
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.

```r
findLocVar function 24c ≡
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 other-
wise.

```r
findFunDef function 24d ≡
findFunDef <- function(fun, cntxt) {
  cenv <- cntxt$env
  info <- findCenvVar(fun, cenv)
}
```r
if (!is.null(info$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

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

```r
findLocals0 <- function(e) {
  if (typeof(e) == "language") {
    if (typeof(e[[1]]) %in% c("symbol", "character"))
      switch(as.character(e[[1]]),
        {findLocals0 switch clauses}
        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.

```r
"=" =,
```n
"<-" = unique(c(getAssignedVar(e), findLocalsList(e[-1]))),

The assigned variable is determined by `getAssignedVar`.

```r
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.

```r
"for" = unique(c(as.character(e[2]), findLocalsList(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.

```r
"delayedAssign" =,
"assign" = if (length(e) == 3 &&
  is.character(e[[2]]) &&
  length(e[[2]]) == 1)
  c(e[[2]], findLocals0(e[[3]], shadowed))
else findLocalsList(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.

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.

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

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

```r
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.

```r
findLocalsList <- function(elist, cntxt) {
  initialShadowedFuns <- c("expression", "local", "quote")
  shadowed <- Filter(function(n) !isBaseVar, 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)
}

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.

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.

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

```
languageFuns <- c("\^", "~", "\<", "\<\-", "\<=", "\<\-", "\="", "\=\=", "\>\>", "\>\=\",
                  "\|", "\|\|", "\|\|", "\|\|", "\|\|", "\|\|", "\|\|", "\|\|", "\|\|", "\|\|",
                  "\{", "\{\", "\$", "\$\<\-", "\$\="", "\$\&\&", "\$\&\&", "\$\%\%", "\$\%\%",
                  "\%\%", "\%\%", "\%\%", "\%\%", "\%\%", "\%\%", "\%\%", "\%\%", "\%\%", "\%\%", "\%\%",
                  "\%\%", "\%\%", "\%\%", "\%\%", "\%\%", "\%\%", "\%\%", "\%\%", "\%\%", "\%\%",
                  "\%\%", "\%\%", "\%\%", "\%\%", "\%\%", "\%\%", "\%\%", "\%\%", "\%\%", "\%\%")
```

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.

```
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.

```r
noInlineSymbols <- c("standardGeneric")

getInlineInfo <- function(name, cntxt) {
  optimize <- cntxt$optimize
  if (optimize > 0 && ! (name %in% noInlineSymbols)) {
    info <- findCenvVar(name, cntxt$env)
    if (is.null(info))
      NULL
    else {
      ftype <- info$ftype
      frame <- info$frame
      if (ftype == "namespace") {
        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.

```r
if (! isNamespace(frame)) {
  ## should be the import frame of the current toplevel
  top <- toplevel(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.

\[ \langle \text{inline handler implementation} \rangle \equiv \]
\[
\text{inlineHandlers} \leftarrow \text{new.env(hash = TRUE, parent = emptyenv())}
\]

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

\[
\text{getInlineHandler} \leftarrow \text{function(name, package = "base")} \{ \\
\text{if (exists(name, inlineHandlers, inherits = FALSE))} \{ \\
\text{hinfo} \leftarrow \text{get(name, inlineHandlers)} \\
\text{if (hinfo$package == package)} \\
\text{hinfo$handler} \\
\text{else NULL} \\
\} \\
\text{else NULL} \\
\}
\]

\[
\text{haveInlineHandler} \leftarrow \text{function(name, package = "base")} \{ \\
\text{if (exists(name, inlineHandlers, inherits = FALSE))} \{ \\
\text{hinfo} \leftarrow \text{get(name, inlineHandlers)} \\
\text{package == hinfo$package} \\
\} \\
\text{else FALSE} \\
\}
\]

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.

\[ \langle \text{install default inlining handlers} \rangle \equiv \]
\[
\text{local} \{ \\
\langle \text{install default SPECIAL handlers} \rangle \\
\langle \text{install default BUILTIN handlers} \rangle \\
\}
\]

The handler installations are wrapped in a \texttt{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 BUILTIN 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.

```r
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`.

```r
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]]
    ## **** handle ... here ??
    cmpBuiltinArgs$<-(arg, names$<-(arg)
    # compile missing BUILTIN argument
    ## signal an error for promise or bytecode argument
    cmpBuiltinArgs$<-(arg
    # compile a general BUILTIN argument
  }
}
```
Missing argument code is generated by
\[
\langle \text{compile missing BUILTIN argument} \rangle \equiv \begin{cases} 
\text{if (missing(a))} & \{ 
\text{if (missingOK)} & \{ 
\text{cb$putcode(DOMISSING.OP)} 
\text{cmpTag(n, cb)} 
\} 
\text{else} & \{ 
\text{cntxt$stop(gettext("missing arguments are not allowed"), cntxt)} 
\} 
\} 
\end{cases}
\]

The error case should not be reached as \texttt{cmpBuiltinArgs} should not be called with missing arguments unless \texttt{missingOK} is \texttt{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 \texttt{cmp} is not reached in this case. Constant folding is needed here since it doesn’t go through \texttt{cmp}.

\[
\langle \text{compile a general BUILTIN argument} \rangle \equiv \begin{cases} 
\text{if (is.symbol(a))} & \{ 
\text{ca <- constantFold(a, cntxt)} 
\text{if (is.null(ca))} & \{ 
\text{cmpSym(a, cb, ncntxt, missingOK)} 
\text{cb$putcode(PUSHARG.OP)} 
\} 
\text{else} & \{ 
\text{cmpConstArg(ca$value, cb, cntxt)} 
\} 
\} 
\text{else if (typeof(a) == "language")} & \{ 
\text{cmp(a, cb, ncntxt)} 
\text{cb$putcode(PUSHARG.OP)} 
\} 
\text{else} & \{ 
\text{cmpConstArg(a, cb, cntxt)} 
\text{cmpTag(n, cb)} 
\} 
\end{cases}
\]

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

Default handlers for all BUILTIN functions in the \texttt{base} package are installed programmatically by
\[
\langle \text{install default BUILTIN handlers} \rangle \equiv \begin{cases} 
\text{for (b in basevars[types == "builtin"])} & \{ 
\text{if (! haveInLineHandler(b, "base"))} & \{ 
\text{setInLineHandler(b, cmpBuiltin)} 
\} 
\} 
\end{cases}
\]

### 7.2 SPECIAL functions

Calls to functions known to be of type SPECIAL can also be compiled somewhat more efficiently by the \texttt{cmpSpecial} function:
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.

```r
setInlineHandler("{", function(e, cb, cntxt) {
  n <- length(e)
  if (n == 1)
    cmp(NULL, cb, cntxt)
  else {
    ncntxt <- make.noValueContext(cntxt)
    for (i in 2 : (n - 1)) {
      cmp(e[[i]], cb, ncntxt)
      cb$putcode(POP.OP)
    }
  }
})
```
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.

```r
⟨inlining handler for function⟩ ≡
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.

\[\text{\textbf{inlining handler for } } (37a) \equiv \text{\textbf{inlining handler for } } (37b) \equiv \]

\[
\text{\textbf{inlining handler for } } (37a) \equiv \text{\textbf{inlining handler for } } (37b) \equiv
\]

\[
\text{\textbf{inlining handler for } } (37a) \equiv \text{\textbf{inlining handler for } } (37b) \equiv
\]

\[
\text{\textbf{inlining handler for } } (37a) \equiv \text{\textbf{inlining handler for } } (37b) \equiv
\]

\[
\text{\textbf{inlining handler for } } (37a) \equiv \text{\textbf{inlining handler for } } (37b) \equiv
\]

\[
\text{\textbf{inlining handler for } } (37a) \equiv \text{\textbf{inlining handler for } } (37b) \equiv
\]

\[
\text{\textbf{inlining handler for } } (37a) \equiv \text{\textbf{inlining handler for } } (37b) \equiv
\]

\[
\text{\textbf{inlining handler for } } (37a) \equiv \text{\textbf{inlining handler for } } (37b) \equiv
\]

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.

\[
\text{\textbf{inlining handler for } } (37a) \equiv \text{\textbf{inlining handler for } } (37b) \equiv
\]

\[
\text{\textbf{inlining handler for } } (37a) \equiv \text{\textbf{inlining handler for } } (37b) \equiv
\]

\[
\text{\textbf{inlining handler for } } (37a) \equiv \text{\textbf{inlining handler for } } (37b) \equiv
\]

\[
\text{\textbf{inlining handler for } } (37a) \equiv \text{\textbf{inlining handler for } } (37b) \equiv
\]

\[
\text{\textbf{inlining handler for } } (37a) \equiv \text{\textbf{inlining handler for } } (37b) \equiv
\]

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

\[
\text{\textbf{local(expr)}}
\]
essentially the same as

\((\text{function}(\text{expr}))(\text{expr})\)

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

\begin{verbatim}
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
})
\end{verbatim}

### 8.6 The return function

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

\begin{verbatim}
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
})
\end{verbatim}
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.

```plaintext
\[\text{if inline handler body}\]
```

\[
\text{test} \leftarrow e[2]
\text{then.expr} \leftarrow e[3]
\text{if } (\text{length}(e) == 4) \{ \\
\quad \text{have.else.expr} \leftarrow \text{TRUE} \\
\quad \text{else.expr} \leftarrow e[4]
\}
\text{else have.else.expr} \leftarrow \text{FALSE}
\]

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 \text{TRUE} or \text{FALSE} then only the appropriate branch is compiled and the handler returns \text{TRUE}.

```plaintext
\[\text{if inline handler body}\]
```

\[
\text{ct} \leftarrow \text{constantFold}(\text{test}, \text{cntxt})
\]

\[
\text{if } (! \text{is.null}(\text{ct}) \&\& \text{is.logical}(\text{ct}\$value) \&\& \text{length}(\text{ct}\$value) == 1 \\
\quad \&\& \! \text{is.na}(\text{ct}\$value)) \{ \\
\quad \text{if } (\text{ct}\$value) \\
\quad \quad \text{cmp}(\text{then.expr}, \text{cb}, \text{cntxt}) \\
\quad \text{else if } (\text{have.else.expr}) \\
\quad \quad \text{cmp}(\text{else.expr}, \text{cb}, \text{cntxt}) \\
\quad \text{else if } (\text{cntxt}\$tailcall) \{ \\
\quad \quad \text{cb}\$putcode(\text{LDNULL.OP}) \\
\quad \quad \text{cb}\$putcode(\text{INVISIBLE.OP}) \\
\quad \quad \text{cb}\$putcode(\text{RETURN.OP}) \\
\quad \} \\
\quad \text{else cb}\$putcode(\text{LDNULL.OP}) \\
\text{return}(\text{TRUE})
\}
\]

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 \text{TRUE}) expression. The \text{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.

```plaintext
ncntxt <- make.nonTailCallContext(cntxt)
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.

```plaintext
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$putcode(INVISIBLE.OP)
    cb$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.

```plaintext
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

```plaintext
setInlineHandler("if", function(e, cb, cntxt) {
  ## **** test for missing, ...
```

```plaintext
40a
defs: inline handler body 39a +≡
40b
defs: inline handler body 39a +≡
40c
defs: inline handler body 39a +≡
40d
defs: inline handler body 39a +≡
40e
defs: inline handler body 39a +≡
40f
defs: inline handler body 39a +≡
40g
defs: inline handler body 39a +≡
40h
defs: inline handler body 39a +≡
40i
defs: inline handler body 39a +≡
40j
defs: inline handler body 39a +≡
40k
defs: inline handler body 39a +≡
40l
defs: inline handler body 39a +≡
40m
defs: inline handler body 39a +≡
40n
defs: inline handler body 39a +≡
40o
defs: inline handler body 39a +≡
40p
defs: inline handler body 39a +≡
40q
defs: inline handler body 39a +≡
40r
defs: inline handler body 39a +≡
40s
defs: inline handler body 39a +≡
40t
defs: inline handler body 39a +≡
40u
defs: inline handler body 39a +≡
40v
defs: inline handler body 39a +≡
40w
defs: inline handler body 39a +≡
40x
defs: inline handler body 39a +≡
40y
defs: inline handler body 39a +≡
40z
defs: inline handler body 39a +≡
```
\section{Inlining handlers for $\&\&$ and $||$ expressions}

In many languages it is possible to convert the expression $a \&\& b$ to an equivalent \texttt{if} expression of the form

\begin{verbatim}
if (a) { if (b) TRUE else FALSE }
\end{verbatim}

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

\begin{verbatim}
if (a) TRUE else if (b) TRUE else FALSE
\end{verbatim}

Compilation of these expressions is thus reduced to compiling \texttt{if} expressions.

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

The code generator for $\&\&$ expressions generates code to evaluate the first argument and then emits an \texttt{AND1ST} instruction. The \texttt{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 \texttt{FALSE} then \texttt{AND1ST} jumps to the label and skips evaluation of the second argument; the value of the expression is \texttt{FALSE}. The code for the second argument is generated next, followed by an \texttt{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 \texttt{RETURN} instruction is generated if the $\&\&$ expression was in tail position.

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

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

\begin{verbatim}
setInlineHandler("||", function(e, cb, cntxt) {
    ## **** arity check??
    ncntxt <- make.argContext(cntxt)
\end{verbatim}
callidx <- cb$putconst(e)
label <- cb$makelabel()
\texttt{cmp}(e[2], cb, ncntxt)
cb$putcode(OR1ST.OP, callidx, label)
\texttt{cmp}(e[3], cb, ncntxt)
cb$putcode(OR2ND.OP, callidx)
cb$putlabel(label)
if (cntxt$tailcall)
  cb$putcode(RETURN.OP)
TRUE
})

10 Loops

In principle code for \texttt{repeat} and \texttt{while} loops can be generated using just \texttt{GOTO} and \texttt{BRIFNOT} instructions. \texttt{for} loops require a little more to manage the loop variable and termination. A complication arises due to the need to support \texttt{break} and \texttt{next} calls in the context of lazy evaluation of arguments: if a \texttt{break} or \texttt{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 \texttt{return} function as described in Section 8.6.

To support these non-local jumps the interpreter sets up a \texttt{setjmp} context for each loop, and \texttt{break} and \texttt{next} use \texttt{longjmp} to transfer control. In general, compiled loops need to use a similar approach. For now, this is implemented by the \texttt{STARTLOOPCNTXT} and \texttt{ENDLOOPCNTXT} instructions. The \texttt{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 \texttt{ENDLOOPCNTXT}.

At least with some assumptions it is often possible to implement \texttt{break} and \texttt{next} calls as simple \texttt{GOTO} s. If all \texttt{break} and \texttt{next} calls in a loop can be implemented using \texttt{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 \texttt{setjmp} per \texttt{STARTLOOPCNTXT} and uses nested calls to \texttt{bceval} to run the code. Eventually we should be able to reduce the need for nested \texttt{bceval} calls and to arrange that \texttt{setjmp} buffers be reused for multiple purposes.

10.1 \texttt{repeat} loops

The simplest loop in R is the \texttt{repeat} loop. The code generator is defined as

\begin{verbatim}
setInlineHandler("repeat", function(e, cb, cntxt) {
  body <- e[[2]]
  \texttt{generate context and body code for repeat loop} \texttt{43a}
  \texttt{generate repeat and while loop wrap-up code} \texttt{43c}
  TRUE
})
\end{verbatim}

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. \texttt{genCode} is used to produce a new code object containing code for the loop body followed by the \texttt{ENDLOOPCNTXT} instruction. The code object is then installed in the context pool. The \texttt{STARTLOOPCNTXT} instruction is emitted to the main code buffer. When a separate code object is written the need for using \texttt{RETURNJMP} for return calls is indicated by setting the \texttt{needRETURNJMP} flag in the compiler context to TRUE.

\begin{verbatim}
43a (generate context and body code for repeat loop) ≡
    \begin{cases}
    \text{genCode(body, cntxt, function(cb, cntxt) { }
    \text{cmpRepeatBody(body, cb, cntxt) \}
    cb$putcode(ENDLOOPCNTXT.OP)}
    \end{cases}

43b (cmpRepeatBody function) ≡
    \begin{cases}
    \text{cmpRepeatBody <- function(body, cb, cntxt) { }
    \text{loop.label <- cb$makelabel()}
    \text{end.label <- cb$makelabel()}
    \text{cb$putlabel(loop.label)}
    \text{lcntxt <- make.loopContext(cntxt, loop.label, end.label)}
    \text{cmp(body, cb, lcntxt)}
    \text{cb$putcode(POP.OP)}
    \text{cb$putcode(GOTO.OP, loop.label)}
    \text{cb$putlabel(end.label)}
    \end{cases}

43c (generate repeat and while loop wrap-up code) ≡
    \begin{cases}
    \text{cb$putcode(LDNULL.OP)}
    \end{cases}
\end{verbatim}
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.

\[
\langle \text{inlining handlers for next and break} \rangle \equiv \\
\text{setInlineHandler}(\text{"break"}, \text{function}(e, cb, cntxt) \{ \\
\quad \text{if (is.null(cntxt$loop))} \{ \\
\quad\text{notifyWrongBreakNext("break", cntxt) } \\
\quad\text{cmpSpecial}(e, cb, cntxt) \\
\quad\} \\
\quad\text{else if (cntxt$loop$gotoOK) } \{ \\
\quad\text{cb$putcode(GOTO.OP, cntxt$loop$end) } \\
\quad\text{TRUE} \\
\quad\} \\
\quad\text{else cmpSpecial}(e, cb, cntxt) \\
\}) \\
\text{setInlineHandler}(\text{"next"}, \text{function}(e, cb, cntxt) \{ \\
\quad \text{if (is.null(cntxt$loop))} \{ \\
\quad\text{notifyWrongBreakNext("next", cntxt) } \\
\quad\text{cmpSpecial}(e, cb, cntxt) \\
\quad\} \\
\quad\text{else if (cntxt$loop$gotoOK) } \{ \\
\quad\text{cb$putcode(GOTO.OP, cntxt$loop$loop) } \\
\quad\text{TRUE} \\
\quad\} \\
\quad\text{else cmpSpecial}(e, cb, cntxt) \\
\})
\]

10.2 while loops

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

\[
\langle \text{inlining handler for while loops} \rangle \equiv \\
\text{setInlineHandler}(\text{"while"}, \text{function}(e, cb, cntxt) \{ \\
\quad \text{cond <- e[[2]]} \\
\quad \text{body <- e[[3]]} \\
\quad \langle \text{generate context and body code for while loop} \rangle \equiv \\
\quad \langle \text{generate repeat and while loop wrap-up code} \rangle \equiv \\
\quad \text{TRUE} \\
\})
\]

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.

\[
\langle \text{generate context and body code for while loop} \rangle \equiv \\
\quad \text{if (checkSkipLoopCntxt(cond, cntxt) && checkSkipLoopCntxt(body, cntxt))} \\
\quad \text{cmpWhileBody(e, cond, body, cb, cntxt)} \\
\]
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 BRIFNOTE 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.

\[\text{cmpWhileBody function}\]

\[\text{cmpWhileBody} <- \text{function}(\text{call}, \text{cond}, \text{body}, \text{cb}, \text{cntxt}) \{\]
\[
    \text{loop.label} <- \text{cb$makelabel()}
    \text{end.label} <- \text{cb$makelabel()}
    \text{cb$putlabel(loop.label)}
    \text{lcntxt} <- \text{make.loopContext}(cntxt, loop.label, end.label)
    \text{cmp}(\text{cond}, \text{cb}, \text{lcntxt})
    \text{callidx} <- \text{cb$putconst(call)}
    \text{cb$putcode(BRIFNOT.OP, callidx, end.label)}
    \text{cmp(body, cb, lcntxt)}
    \text{cb$putcode(POP.OP)}
    \text{cb$putcode(GOTO.OP, loop.label)}
    \text{cb$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 to 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⟩
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⟩
    ⟨generate for loop wrap-up code⟩
    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⟩
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 \texttt{STEPFOR} instruction. This is labeled by the \texttt{loop.label} label, since this will also be the target used by a \texttt{next} expression. An additional label, \texttt{body.label} is needed for the top of the loop, which is used by \texttt{STEPFOR} if there are more loop elements to process. When the \texttt{ci} argument is not NULL code is being generated for the case without a \texttt{setjmp} context, and the first instruction is the \texttt{STARTFOR} instruction which initializes the loop and jumps to \texttt{loop.label} at the \texttt{STEPLOOP} instruction.

\begin{lstlisting}[language=R]
\texttt{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)
  \texttt{cmp(body, cb, lcntxt)}
  cb$putcode(POP.OP)
  cb$putlabel(loop.label)
  cb$putcode(STEPFOR.OP, body.label)
  cb$putlabel(end.label)
}
\end{lstlisting}

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

\begin{lstlisting}[language=R]
\texttt{generate for loop wrap-up code} \equiv
  cb$putcode(ENDFOR.OP)
  if (cntxt$tailcall) {
    cb$putcode(INVOKE.OP)
    cb$putcode(RETURN.OP)
  }
\end{lstlisting}

\section{Avoiding runtime loop contexts}

When all uses of \texttt{break} or \texttt{next} in a loop occur only in top level contexts then all \texttt{break} and \texttt{next} calls can be implemented with simple GOTO instructions and a \texttt{setjmp} context for the loop is not needed. Top level contexts are the loop body itself and argument expressions in top level calls to \texttt{if}, \{, and \}. The \texttt{switch} functions will eventually be included as well. The function \texttt{checkSkipLoopContxt} recursively traverses an expression tree to determine whether all relevant uses of \texttt{break} or \texttt{next} are safe to compile as GOTO instructions. The search returns \texttt{FALSE} if a \texttt{break} or \texttt{next} call occurs in an unsafe place. The search stops and returns \texttt{TRUE} for any expression that cannot contain relevant \texttt{break} or \texttt{next} calls. These stop expressions are calls to the three loop functions and to \texttt{function}. Calls to functions like \texttt{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

\[\text{checkSkipLoopCntxt} \leftarrow \text{function}(e, \text{cntxt}, \text{breakOK} = \text{TRUE}) \{
\text{if} (\text{typeof}(e) == \"language\") \{
\text{fun} \leftarrow e[[1]]
\text{if} (\text{typeof(fun)} == \"symbol\") \{
\text{fname} \leftarrow \text{as.character(fun)}
\text{if} (! \text{breakOK} \&\& \text{fname} \%in\% c("break", "next"))
\text{FALSE}
\text{else if} (\text{isLoopStopFun(fname, cntxt)})
\text{TRUE}
\text{else if} (\text{isLoopTopFun(fname, cntxt)})
\text{checkSkipLoopCntxtList}(e[-1], \text{cntxt}, \text{breakOK})
\text{else}
\text{checkSkipLoopCntxtList}(e[-1], \text{cntxt}, \text{FALSE})
\}
\text{else}
\text{checkSkipLoopCntxtList}(e, \text{cntxt}, \text{FALSE})
\}
\text{else TRUE}
\}
\]

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

\[\text{checkSkipLoopCntxtList} \leftarrow \text{function}(\text{elist}, \text{cntxt}, \text{breakOK}) \{
\text{for} (a \text{ in as.list(elist)})
\text{if} (! \text{missing(a)} \&\& ! \text{checkSkipLoopCntxt(a, cntxt, breakOK)})
\text{return}(\text{FALSE})
\}
\text{TRUE}
\]

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

\[\text{isLoopStopFun} \leftarrow \text{function}(\text{fname, cntxt}) \{
(\text{fname} \%in\% c("function", "for", "while", "repeat") \&\&
\text{isBaseVar(fname, cntxt)})
\}
\]

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

\[\text{isLoopTopFun} \leftarrow \text{function}(\text{fname, cntxt}) \{
(\text{fname} \%in\% c("", ",", "if") \&\&
\text{isBaseVar(fname, cntxt)})
\}
\]

The \text{checkSkipLoopCntxt} function does not check whether calls to \text{break} or \text{next} are indeed calls to the \text{base} functions. Given the special syntactic nature of \text{break} and \text{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 \( \times \) and \(\div\) 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

\[
\text{setInlineHandler}(\text{"+"}, \text{function}(e, cb, cntxt) \{
\text{if (length(e) == 3)}
\text{cmpPrim2(e, cb, ADD.OP, cntxt)}
\text{else}
\text{cmpPrim1(e, cb, UPLUS.OP, cntxt)}
\})
\]

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

The code generators for multiplication and division are

\[
\text{setInlineHandler}(\text{"\times"}, \text{function}(e, cb, cntxt) \{
\text{cmpPrim2(e, cb, MUL.OP, cntxt)})
\]

\[
\text{setInlineHandler}(\text{"\div"}, \text{function}(e, cb, cntxt) \{
\text{cmpPrim2(e, cb, DIV.OP, cntxt))}
\]

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 there are any missing or \(\ldots\) 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.

\[
\text{cmpPrim1 <- function}(e, cb, op, cntxt) \{
\text{if (dots.or.missing(e[-1]))}
\text{cmpBuiltin(e, cb, cntxt)}
\text{else if (length(e) != 2) \{ }
\text{notifyWrongArgCount(e[[1]], cntxt)}
\text{cmpBuiltin(e, cb, cntxt)}
\text{\}}
\]

\[
\text{else \{ }
\text{ntcntxt <- make.nonTailCallContext(cntxt)}
\]
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.

```
cmp(e[2], 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:

```
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:

```
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 default inline handler action is therefore to use cmpSpecial. For calls with one unnamed argument the LOG.OP instruction is used. For two unnamed arguments LOGBASE.OP is used. It might be useful to introduce instructions for log2 and log10 as well but this has not been done yet.
A number of one argument math functions are handled by the interpreter using the function `math1` in `arithmetic.c`. The MATH1.OP instruction handles these for compiled code. The instruction takes two operands, an index for the call expression in the constant table, and an index for the function to be called in a table of function pointers. The table of names in the byte code compiler has to match the function pointer array in the byte code interpreter. It would have been possible to use the same indices as the offset values used in `names.c`, but keeping this consistent seemed more challenging.

The code generation is done by `cmpMath1`.

```r
## Keep the order consistent with the order in the internal byte code interpreter!
```

The code generation is done by `cmpMath1`.

```r
cmpMath1 <- function(e, cb, cntxt) {
  if (dots.or.missing(e[-1]))
    cmpBuiltin(e[-1], cb, cntxt)
  else if (length(e) != 2) {
    notifyWrongArgCount(e[[1]], cntxt)
    cmpBuiltin(e[[1]], cb, cntxt)
  }
}
```
else {
    name <- as.character(e[[1]])
    idx <- match(name, math1funs) - 1
    if (is.na(idx))
        stop(sQuote(name), "is not a registered math1 function")
    ncntxt <- make.nonTailCallContext(cntxt)
    cmp(e[[2]], cb, ncntxt);
    ci <- cb$putconst(e)
    cb$putcode(MATH1.OP, ci, idx)
    if (cntxt$tailcall)
        cb$putcode(RETURN.OP)
    TRUE
}

The generators are installed by

\[ \text{inline one argument math functions} \]
\[ \forall \text{name in math1funs} \]
\[ \text{setInlineHandler(name, cmpMath1)} \]

11.2 Logical operators

Two argument instructions are provided for the comparison operators and code for them can be
generated using \( \text{cmpPrim2} \):

\[ \text{inline handlers for comparison operators} \]
\[ \forall \text{name in math1funs} \]
\[ \text{setInlineHandler(name, function(e, cb, cntxt) \{ cmpPrim2(e, cb, EQ.OP, cntxt) \}}} \]

\[ \text{setInlineHandler("!="), function(e, cb, cntxt) \{ cmpPrim2(e, cb, NE.OP, cntxt) \}} \]

\[ \text{setInlineHandler("<"), function(e, cb, cntxt) \{ cmpPrim2(e, cb, LT.OP, cntxt) \}} \]

\[ \text{setInlineHandler("<="), function(e, cb, cntxt) \{ cmpPrim2(e, cb, LE.OP, cntxt) \}} \]

\[ \text{setInlineHandler(">"), function(e, cb, cntxt) \{ cmpPrim2(e, cb, GT.OP, cntxt) \}} \]

The vectorized & and | functions are handled similarly:

\[ \text{inline handlers for \& and |} \]
\[ \forall \text{name in math1funs} \]
\[ \text{setInlineHandler("&", function(e, cb, cntxt) \{ cmpPrim2(e, cb, AND.OP, cntxt) \}} \]
The negation operator ! takes only one argument and code for calls to it are generated using

```r
setInlineHandler("!", function(e, cb, cntxt)
  cmpPrim1(e, cb, NOT.OP, cntxt))
```

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

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

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

\[\textit{inlining handler for \$} \equiv \textit{SetInlineHandler}^{\textit{\$}}\ (\text{function}(e, cb, cntxt) \{ \\
\quad \textbf{if} \ \text{any.dots}(e) \ || \ \text{length}(e) \neq 3) \\
\quad \text{cmpSpecial}(e, cb, cntxt) \\
\quad \textbf{else} \ { \\
\quad \quad \text{sym} \leftarrow \text{if} \ \text{is.character}(e[[3]]) \\
\quad \quad \quad \text{as.name}(e[[3]]) \ \text{else} \ e[[3]] \\
\quad \quad \textbf{if} \ \text{is.name}(\text{sym}) \ { \\
\quad \quad \quad \text{nctx} \leftarrow \text{make.argContext}(cntxt) \\
\quad \quad \quad \text{cmp}(e[[2]], cb, nctx) \\
\quad \quad \quad \text{ci} \leftarrow \text{cb$putconst}(e) \\
\quad \quad \quad \text{csi} \leftarrow \text{cb$putconst}(\text{sym}) \\
\quad \quad \quad \text{cb$putcode(DOLLAR.OP, ci, csi)} \\
\quad \quad \quad \textbf{if} \ \text{cntxt$tailcall} \ \text{cb$putcode(RETURN.OP)} \\
\quad \quad \quad \text{TRUE} \\
\quad \quad \textbf{} \\
\quad \quad \textbf{else} \ \text{cmpSpecial}(e, cb, cntxt) \\
\quad \textbf{} \\
\text{\}} \\
\text{\}} \]
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

```r
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 does not depend on the position of its calls in the evaluation stack. As a result, a call of the form

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

can be replaced by the call

```r
.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

```r
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`:

```r
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
}
}

\textit{inline safe simple .Internal functions from base}\quad \equiv
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)

\textit{inline safe simple .Internal functions from stats}\quad \equiv
SafeStatsInternals <- c("dbinom", "dcauchy", "dgeom", "dhyper", "dlnorm",
"dlogis", "dnorm", "dpois", "dnunif", "dweibull",
"fft", "mvfft", "pbeta", "pcauchy",
"pgamma", "phyper", "plnorm", "plogis", "pnorm",
"ppois", "punif", "pweibull", "qbinom", "qcauchy",
"qgamma", "qhyper", "qlnorm", "qlogis", "qnorm",
"qpois", "qunif", "queibull", "rbinom", "rcauchy",
"rgeom", "rhyper", "rlnorm", "rlogis", "rnorm",
"rpois", "rsignrank", "runif", "rweibull",
"rwilcox", "ptukey")

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 \texttt{BUILTIN}, and the function \texttt{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 \texttt{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, \texttt{simpleInternals} is useful for providing a list of candidates to screen. The \texttt{is.simpleInternal} function can be used in test code to check that the assumption made in the compiler is valid. The implementation is

\textit{simpleInternals function}\quad \equiv

\begin{verbatim}
simpleInternals <- function(pos = "package:base") {
  names <- ls(pos = pos, all = T)
  if (length(names) == 0)
    character(0)
  }
\end{verbatim}
else {
    fn <- function(n)
        is.simpleInternal(get(n, pos = pos))
    names[sapply(names, fn)]
}
```r
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 BUILTINs 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.

```r
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

```r
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 BUILTINS. It might be worth defining virtual machine instructions for them as well.

11.6 Inline handler for calling C functions

The `.Call` interface is now the preferred interface for calling C functions and is also used in base packages like `stat`. The `DOTCALL.OP` instruction allows these calls to be made without allocating a list of arguments—the arguments are accumulated on the stack. For now only 16 or fewer arguments are handled; more arguments, and cases with named arguments, are handled by the standard `.Call BUILTIN`.

```
⟨inline handler for .Call⟩ ≡ (102b)
setInlineHandler(".Call", function(e, cb, cntxt) {
  nargsmax <- 16  ## should match DOTCALL_MAX in eval.c
  if (dots.or.missing(e[-1]) || !is.null(names(e)) ||
      length(e) < 2 || length(e) > nargsmax + 2)
    cmpBuiltin(e, cb, cntxt)  ## punt
  else {
    ncntxt <- make.nonTailCallContext(cntxt)
    cmp(e[[2]], cb, ncntxt);
    nargs <- length(e) - 2
    if (nargs > 0) {
      ncntxt <- make.argContext(cntxt)
      for (a in as.list(e[-(1:2)]))
        cmp(a, cb, ncntxt);
    }
    ci <- cb$putconst(e)
    cb$putcode(DOTCALL.OP, ci, nargs)
    if (cntxt$tailcall)
      cb$putcode(RETURN.OP)
    TRUE
  }
})
```

11.7 Inline handlers for generating integer sequences

The colon operator and the BUILTIN functions `seq_along` and `seq_len` generate sequences (the sequence might not be integers if long vectors are involved or the colon operator is given no-integer arguments). The `COLON.OP`, `SEQALONG.OP`, and `SEQLEN.OP` instructions implement these operations in byte code. This allows an implementation in which the result stored on the stack is not a fully realized sequence but only a recipe that the `for` loop, for example, can use to run the loop without generating the sequence. This is optionally implemented in the byte code interpreter. It would also be possible to allow the compact sequence representation to be stored in variables, etc., but this would require more extensive changes.

```
⟨inline handlers for integer sequences⟩ ≡ (102b)
setInlineHandler("::", function(e, cb, cntxt)
```

11.8 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`.

```
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

```
setInlineHandler("::", function(e, cb, cntxt)
  cmpMultiColon(e, cb, SEQALONG.OP, cntxt))
setInlineHandler(":::", function(e, cb, cntxt)
  cmpMultiColon(e, cb, SEQLEN.OP, cntxt))
```

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.

```
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.

```javascript
function(e, cb, ctxt) {
    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.

```javascript
function(e, cb, ctxt) {
    if (length(e) < 2 || any.dots(e))
        cmpSpecial(e, cb, ctxt)
    else {
        ## **** check name on EXPR, if any, partially matches EXPR?
        ## collect information on named alternatives
        ## create the labels
        ## create the map from names to labels for a character switch
        ## emit code for the EXPR argument
        ## emit the switch instruction
        ## emit error code for empty alternative in numerical switch
```
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`. A warning is issued if there are no cases. 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.

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.

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.
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.

⟨create the labels⟩ ≡ (61b)

## 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.

⟨create the map from names to labels for a character switch⟩ ≡ (61b)

## 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 (! 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 \texttt{findActionIndex}.

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

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.

\begin{verbatim}
## compile the EXPR argument
ncntxt <- make.nonTailCallContext(cntxt)
\end{verbatim}

The switch instruction takes the selector off the stack and four operands from 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 \texttt{patchlables} function of the code buffer.

\begin{verbatim}
## 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)
}
\end{verbatim}

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

## 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

\[ f(x) \leftarrow v \]

or

\[ g(f(x)) \leftarrow v \]

The second form is sometimes called a nested complex assignment. Ordinary assignment creates or modifies a binding in the current environment. Superassignment via the \( \leftarrow \) 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.

```r
## cmpAssign function
cmpAssign <- function(e, cb, cntxt) {
  if (! checkAssign(e, cntxt))
    return(cmpSpecial(e, cb, cntxt))
  superAssign <- as.character(e[[1]]) == "\leftarrow"
  lhs <- e[[2]]
  value <- e[[3]]
  return(cmpAssign(lhs, value, superAssign, cntxt))
}
```
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

```r
setInlineHandler("-", cmpAssign)
setInlineHandler("=", cmpAssign)
setInlineHandler("<<-", cmpAssign)
```

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

```r
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`
    }
  }
}
```

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.

```r
`check left hand side call`
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")
```
### 13.1 Simple assignment expressions

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

```r
⟨cmpSymbolAssign function 67a⟩ ≡ (102b)
cmpSymbolAssign <- function(symbol, value, superAssign, cb, cntxt) {
  ⟨compile the right hand side value expression 67b⟩
  ⟨emit code for the symbol assignment instruction 67c⟩
  ⟨for tail calls return the value invisibly 67d⟩
  TRUE
}
```

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

```r
⟨compile the right hand side value expression 67b⟩ ≡ (67a 69b)
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.

```r
⟨emit code for the symbol assignment instruction 67c⟩ ≡ (67a)
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`.

```r
⟨for tail calls return the value invisibly 67d⟩ ≡ (67a 69b)
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:

```c
/*
 * 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:

```
inner assignment trashes temporary ≡
  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:

```r
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 interpreter. 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

```r
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`

```r
cmpComplexAssign <- function(symbol, lhs, value, superAssign, cb, cntxt) {
  (select complex assignment instructions)
  (compile the right hand side value expression)
  (compile the left hand side call)
  (for tail calls return the value invisibly)
  TRUE;
}
```

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

```r
compile the left hand side call
csi <- cb$putconst(symbol)
```

```r
compile code to compute left hand side values
```

```r
compile code to compute right hand side values
```
The appropriate instructions \texttt{startOP} and \texttt{endOP} depend on whether the assignment is an ordinary assignment or a superassignment.

\begin{verbatim}
if (superAssign) {
  startOP <- STARTASSIGN2.OP
  endOP <- ENDASSIGN2.OP
}
else {
  if (! findVar(symbol, cntxt))
    notifyUndefVar(symbol, cntxt)
  startOP <- STARTASSIGN.OP
  endOP <- ENDASSIGN.OP
}
\end{verbatim}

An undefined variable notification is issued for ordinary assignment, since this will produce a runtime error. For superassignment \texttt{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 \texttt{flattenPlace} returns a list of the expressions that need to be considered, with \texttt{*tmp*} in place of the current left hand side argument. For example, for an assignment of the form \( f(g(h(x, k), j), i) \leftarrow v \) this produces

\begin{verbatim}
> flattenPlace(quote(f(g(h(x, k), j), i)))
{\Tt{}1\wendquote}
f(''*tmp*'', i)
{\Tt{}2\wendquote}
g(''*tmp*'', j)
{\Tt{}3\wendquote}
h(''*tmp*'', k)
\end{verbatim}

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 \texttt{flattenPlace}, evaluated in reverse order. An argument context is used since there are already values on the stack.

\begin{verbatim}
ncntxt <- make_argContext(cntxt)
flatPlace <- flattenPlace(lhs)
for (p in rev(flatPlace[-1]))
    cmpGetterCall(p, cb, ncntxt)
\end{verbatim}

The compilation of the individual calls carried out by \texttt{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 \texttt{flattenPlace} is defined as

\begin{verbatim}
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
}
\end{verbatim}

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

\begin{verbatim}
cmpSetterCall(flatPlace[[1]], value, cb, ncntxt)
for (p in flatPlace[-1])
    cmpSetterCall(p, as.name("*vtmp*"), cb, ncntxt)
\end{verbatim}

The first call uses the expression for the original right hand side in its call; all others will use \texttt{*vtmp*}. Each replacement function call compiled by \texttt{cmpSetterCall} will remove the top two elements from the stack and then push the new right hand side value on the stack. \texttt{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 \texttt{$<$-}, 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 \texttt{foo::bar} or \texttt{foo:::bar} differs only compiling the function expression and using CHECKFUN to verify the result and prepare the stack.

\begin{verbatim}
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$putcode(GETFUN.OP, ci)
      cmpCallArgs(place[-c(1, 2)], cb, ncntxt)
      cci <- cb$putconst(acall)
      cvi <- cb$putconst(vexpr)
      cb$putcode(SETTER_CALL.OP, cci, cvi)
    }
  }
  else {
    cmp(afun, cb, ncntxt)
    cb$putcode(CHECKFUN.OP)
  }
}
\end{verbatim}

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

\begin{verbatim}
cmpCallArgs(place[-c(1, 2)], cb, ncntxt)
cci <- cb$putconst(acall)
 cvi <- cb$putconst(vexpr)
 cb$putcode(SETTER_CALL.OP, cci, cvi)
\end{verbatim}

The PUSHNULL instruction places \texttt{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 \texttt{fun} is computed by \texttt{getAssignFun}. If \texttt{fun} is a symbol then the assignment function is the symbol followed by \texttt{-}. The function \texttt{fun} can also be an expression of the form \texttt{foo:::bar}, in which case the replacement function is the expression \texttt{foo:::'bar<-'.} NULL is returned if \texttt{fun} does not fit into one of these two cases.

\begin{verbatim}
getAssignFun <- function(fun) {
  if (typeof(fun) == "symbol")
\end{verbatim}
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.

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.

\[
\text{setter inline handler for } $<- \text{\rangle} \equiv (102b)
\]

\[
\text{setter inline handler for } $<- \langle
\setSetErrorHandler("$<-", \text{function(afun, place, call, cb, cntxt) \{\}
    \text{if } \text{any.dots}(place) \text{ || length(place) } \neq 3)
    \text{FALSE}
    \text{else \{\}
        \text{sym } <- \text{place}[3]\n        \text{if } (\text{is.character(sym)})
            \text{sym } <- \text{as.name(sym)}
        \text{else if } (\text{is.name(sym)}) \{\}
            \text{ci } <- \text{cb$putconst(call)}
            \text{csi } <- \text{cb$putconst(sym)}
            \text{cb$putcode(DOLLARGETS.OP, ci, csi)}
            \text{TRUE}
        \}
    \text{else FALSE}
    \}
\}
\]

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

\[
\text{cmpSetterDispatch function} \langle \text{cmpSetterDispatch} \langle\]

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

\[
\langle \text{setter inline handlers for } [<- \text{ and } [[<- \rangle \equiv (102b)
\]

# **** this is now handled differently; see "Improved subset ..."
# 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.

\[
\langle \text{setter inlining handler for } @<- \rangle \equiv (102b)
\]

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
})

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.

\[
\langle \text{cmpGetterCall function} \rangle \equiv (102b)
\]

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)
      \langle compile additional arguments and call to getter function \rangle
    }
  } else {
  }
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.

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.
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.

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.
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

\[ \text{getter inline handlers for } [ \text{ and } [[ ]]} \equiv \]

\[
\text{**** this is now handled differently; see "Improved subset ..."}
\text{setGetterInlineHandler("[", function(call, cb, cntxt)
  \text{cmpGetterDispatch(STARTSUBSET.OP, DFLTSUBSET.OP, call, cb, cntxt))}
\text{setGetterInlineHandler("[[", function(call, cb, cntxt)
  \text{cmpGetterDispatch(STARTSUBSET2.OP, DFLTSUBSET2.OP, call, cb, cntxt))}
\]

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

\[ 1 / \sqrt{2 \times \pi} \]

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 \text{constantFold} function examines its expression argument and handles each expression type by calling an appropriate function.

\[ \text{constantFold function} \equiv \]

\[
\text{**** rewrite using switch??}
\text{constantFold} <- function(e, cntxt) {
  type = typeof(e)
  if (type == "language")
    constantFoldCall(e, cntxt)
  else if (type == "symbol")
    constantFoldSym(e, cntxt)
\]
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.

```r
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

```r
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.

```r
constantFoldSym <- function(var, cntxt) {
  var <- as.character(var)
  if (var %in% constNames && isBaseVar(var, cntxt))
    checkConst(get(var, .BaseNamespaceEnv))
  else
    NULL
}
```

```r
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.

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

The functions in the base package eligible for constant folding are

\[
\text{foldFuns definition} \equiv \begin{array}{c}
\text{foldFuns } \leftarrow \text{\{"+", ",\", "/", ",\", ",\", ",\", ",\", ",\", ",\", ",\", ",\", \text{abs}, \text{acos}, \text{acosh}, \text{asin}, \text{asinh}, \text{atan}, \text{atan2}, \\
\text{atanh}, \text{ceiling}, \text{choose}, \text{cos}, \text{cosh}, \text{exp}, \text{expm1}, \\
\text{floor}, \text{gamma}, \text{1beta}, \text{1choose}, \text{1gamma}, \text{log}, \text{log10}, \\
\text{log1p}, \text{log2}, \text{max}, \text{min}, \text{prod}, \text{range}, \text{round}, \\
\text{seq.along}, \text{seq.int}, \text{seq.len}, \text{sign}, \text{signif}, \\
\text{sin}, \text{sinh}, \text{sqrt}, \text{sum}, \text{tan}, \text{tanh}, \text{trunc}, \\
\text{baseenv}, \text{emptyenv}, \text{globalenv}, \\
\text{Arg}, \text{Conj}, \text{Im}, \text{Mod}, \text{Re}, \\
\text{is.R}\}\end{array}
\]

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.

getFoldFun function \[
\text{getFoldFun function} \equiv \begin{array}{c}
\text{getFoldFun } \leftarrow \text{function(e, cntxt) } \{ \\
\quad \text{ffun } \leftarrow \text{getFoldFun(fun, cntxt)} \\
\quad \text{if (! is.null(ffun)) } \{ \\
\quad \quad \text{args } \leftarrow \text{as.list(e[-1])} \\
\quad \quad \text{for (i in seq.along(args)) } \{ \\
\quad \quad \quad \text{a } \leftarrow \text{args[i]} \\
\quad \quad \quad \text{if (missing(a))} \\
\quad \quad \quad \quad \text{return(NULL)} \\
\quad \quad \quad \text{val } \leftarrow \text{constantFold(a, cntxt)} \\
\quad \quad \quad \text{if (! is.null(val))} \\
\quad \quad \quad \quad \text{args[i]} \leftarrow \text{list(val$value)} \# \text{**** in case value is NULL} \\
\quad \quad \quad \quad \text{else return(NULL)} \\
\quad \quad \} \\
\quad \quad \text{modes } \leftarrow \text{unlist(lapply(args, mode))} \\
\quad \quad \text{if (all(modes %in% constModes)) } \{ \\
\quad \quad \quad \text{tryCatch(checkConst(do.call(ffun, args)),} \\
\quad \quad \quad \quad \text{error = function(e) NULL) \# \text{**** issue warning??} } \\
\quad \quad \} \\
\quad \quad \text{else NULL} \\
\quad \} \\
\quad \text{else NULL} \\
\quad \text{else NULL} \\
\text{else NULL} \\
\}
\]

}
### For now assume all foldable functions are in base

```r
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.

```r
S1a
<cmpfun function S1a>

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 (type == "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.

```r
S1b
<tryCmpfun function S1b>

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`.

```r
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")

  forms <- parse(infile)
  nforms <- length(forms)
  if (nforms > 0) {
    expr.needed <- 1000
    expr.old <- getOption("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.
As a precaution it is useful to check that `infile` and `outfile` are not the same and signal an error if they are.

```r
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.

```r
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.

```r
cmpframe <- function(inpos, file) {
  expr.needed <- 1000
  expr.old <- getOption("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 = "")
}

\texttt{cmplib} \equiv \texttt{cmpfun}

\begin{verbatim}
\texttt{cmpplib <- 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)
}}
\end{verbatim}

15.3 Enabling implicit compilation

\begin{verbatim}
\texttt{enableJIT} \equiv \texttt{cmpfun}

\texttt{enableJIT <- function(level)
    .Internal(enableJIT(level))}
\end{verbatim}

\begin{verbatim}
\texttt{compilePKGS} \equiv \texttt{cmpfun}

\texttt{compilePKGS <- function(enable)
    .Internal(compilePKGS(enable))}
\end{verbatim}

15.4 Setting compiler options

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

\begin{verbatim}
\texttt{setCompilerOptions} \equiv \texttt{cmpfun}

\texttt{setCompilerOptions <- function(...) {
    options <- list(...)  
    nm <- names(options)
    for (n in nm)
        if (!exists(n, \texttt{compilerOptions}))
            stop(gettextf("%s is not a valid compiler option", n),
                 domain = NA)
    old <- list()
    newOptions <- \texttt{compilerOptions}
}}
\end{verbatim}
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$optimize))
                newOptions$optimize <- op
            }
        },
        suppressAll = {
            if (identical(op, TRUE) || identical(op, FALSE)) {
                old <- c(old, list(suppressAll = compilerOptions$suppressAll))
                newOptions$suppressAll <- op
            }
        },
        suppressUndefined = {
            if (identical(op, TRUE) || identical(op, FALSE) || is.character(op)) {
                old <- c(old, list(suppressUndefined = compilerOptions$suppressUndefined))
                newOptions$suppressUndefined <- op
            }
        }),
    }
}

jitEnabled <- enableJIT(-1)
checkCompilerOptions(jitEnabled, newOptions)
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.

```r
.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)
        }, {
            message("Error setting optimization level")
        })
}```
When \texttt{enableJIT} is set to 3, loops should be compiled before executing. However, if the \texttt{optimize} option is set to 0 or 1, a compiled loop will call to the same primitive function as is used by the AST interpreter (e.g. \texttt{do...for}), and the compilation would run into infinite recursion. \texttt{checkCompilerOptions} will detect invalid combinations of \texttt{enableJIT} and \texttt{optimize} and report a warning.

\begin{verbatim}
checkCompilerOptions <- function(jitEnabled, options = NULL) {
  optimize <- getCompilerOption("optimize", options)
  if (jitEnabled <= 2 || optimize >= 2)
    TRUE
  else {
    stop(gettextf(
      "invalid compiler options: optimize(==%d)<2 and jitEnabled(==%d)>2",
      optimize, jitEnabled))
    FALSE
  }
}
\end{verbatim}

\subsection{Disassembler}

A minimal disassembler is provided by \texttt{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.

\begin{verbatim}
disassemble <- function(code) {
  .CodeSym <- as.name(".Code")
  .CodeSym <- function(x)
    if (typeof(x) == "list" && length(x) > 0 && identical(x[[1]], .CodeSym))
      disasm(const <- function(x)
        if (typeof(x) == "list" && length(x) > 0 && identical(x[[1]], .CodeSym))
          disasm(x) else x
        disasm(const <- 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 \texttt{.Internal} function \texttt{disassemble} extracts the numeric code vector and constant pool. The function \texttt{bcDecode} uses the \texttt{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.

\[ \langle \text{bcDecode function} \rangle \equiv \]
\[
\text{bcDecode} <- \text{function}(\text{code}) \{
    n <- \text{length}(\text{code})
    \text{ncode} <- \text{vector}(\text{"list"}, n)
    \text{ncode}[1] <- \text{code}[1] \# \text{version number}
    i <- 2
    \text{while} (i <= n) {
        \text{name} <- \text{Opcodes.names}[[\text{code}[i]+1]]
        \text{argc} <- \text{Opcodes.argc}[[\text{code}[i]+1]]
        \text{ncode}[[i]] <- \text{as.name}((\text{name}))
        i <- i+1
        \text{if} (\text{argc} > 0)
            \text{for} (j in 1:\text{argc}) {
                \text{ncode}[[i]] <- \text{code}[i]
                i <- i+1
            }
    }
    \text{ncode}
\}

16 Improved subset and sub-assignment handling

This section describes changes that allow subset and subassign operations to inmost case be handled without allocating list of the index arguments — the arguments are passed on the stack instead. The function \text{cmpSubsetDispatch} is analogous to \text{cmpDispatch} described above. The \text{dflt.op} argument passed information about the instruction to be emitted. For instructions designed for a particular number of arguments the \text{rank} component is \text{FALSE} and no index count is emitted; this is used for \text{VECSUBSET.OP} and \text{MATSUBSET.OP} instructions. If \text{rank} is \text{TRUE}, then the number of indices is emitted as an operand; this is used by the \text{SUBSET.N.OP} instruction.

\[ \langle \text{cmpSubsetDispatch function} \rangle \equiv \]
\[
\text{cmpSubsetDispatch} <- \text{function}(\text{start.op}, \text{dflt.op}, e, \text{cb}, \text{cntxt}) \{
    \text{if} (\text{dists.or.missing}(\text{e}) || ! \text{is.null}(\text{names}(\text{e})) || \text{length}(\text{e}) < 3)
        \text{cntxt}$\text{stop}$(\text{gettext}(\"cannot compile this expression\"), \text{cntxt})
    \text{else} \{
        oe <- \text{e}[2]
        \text{if} (\text{missing}(\text{oe}))
            \text{cntxt}$\text{stop}$(\text{gettext}(\"cannot compile this expression\"), \text{cntxt})
        \text{ncntxt} <- \text{make.argContext}(\text{cntxt})
        \text{ci} <- \text{cb$putconst}(\text{e})
        \text{label} <- \text{cb$makelabel()}$
        \text{cmp}(\text{oe}, \text{cb}, \text{ncntxt})
        \text{cb$putcode}(\text{start.op}, \text{ci}, \text{label})
        \text{indices} <- \text{e}[-c(1,2)]
        \text{for} (i \text{ in seq.along}(\text{indices}))
    \}
\}
The subsetting handlers fall back to using `cmpDispatch` if there are any named arguments or if an error would need to be signaled (we could issue a compiler warning at this point as well). If all arguments are unnamed and there are no dots then `cmpSubsetDispatch` is used; the instruction emitted depends on the argument count.

```
inline handlers for subsetting
```

```
setInlineHandler("[", function(e, cb, cntxt) {
  if (dots.or.missing(e) || !is.null(names(e)) || length(e) < 3)
    cmpDispatch(STARTSUBSET.OP, DFLTSUBSET.OP, e, cb, cntxt) ## punt
  else {
    nidx <- length(e) - 2;
    if (nidx == 1)
      dflt.op <- list(code = VECSUBSET.OP, rank = FALSE)
    else if (nidx == 2)
      dflt.op <- list(code = MATSUBSET.OP, rank = FALSE)
    else
      dflt.op <- list(code = SUBSET_N.OP, rank = TRUE)
    cmpSubsetDispatch(STARTSUBSET_N.OP, dflt.op, e, cb, cntxt)
  }
})
```

Similarly, `cmpSubassignDispatch` is a variant of `cmpSetterDispatch` that passes index arguments on the stack and emits an index count if necessary.

```
cmpSubassignDispatch <- function(start.op, dflt.op, afun, place, call, cb, cntxt) {
```
if (dots.or.missing(place) || ! is.null(names(place)) || length(place) < 3)
cntxt$stop(gettext("cannot compile this expression"), cntxt)
else {
  ci <- cb$putconst(call)
  label <- cb$makelabel()
  cb$putcode(start.op, ci, label)
  indices <- place[-c(1, 2)]
  for (i in seq_along(indices))
    cmp(indices[[i]], cb, cntxt, TRUE)
  if (dflt.op$rank) cb$putcode(dflt.op$code, ci, length(indices))
  else cb$putcode(dflt.op$code, ci)
  cb$putlabel(label)
  TRUE
}

Again the handlers fall back to `cmpSetterDispatch` if there are named arguments or other complication.

\[ inline handlers for subassignment \]
\[ 80 \equiv \]

```r
setSetterInlineHandler("[[<-", function(afun, place, call, cb, cntxt) {
  if (dots.or.missing(place) || ! is.null(names(place)) || length(place) < 3)
    cmpSetterDispatch(STARTSUBASSIGN2.OP, DFLTSUBASSIGN2.OP,
      afun, place, call, cb, cntxt) ## punt
  else {
    nidx <- length(place) - 2
    if (nidx == 1)
      dflt.op <- list(code = VECSUBASSIGN2.OP, rank = FALSE)
    else if (nidx == 2)
      dflt.op <- list(code = MATSUBASSIGN2.OP, rank = FALSE)
    else
      dflt.op <- list(code = SUBASSIGN_N.OP, rank = TRUE)
    cmpSubassignDispatch(STARTSUBASSIGN2.OP, dflt.op, afun, place, call,
      cb, cntxt)
  }
})
```

```r
setSetterInlineHandler("[<-", function(afun, place, call, cb, cntxt) {
  if (dots.or.missing(place) || ! is.null(names(place)) || length(place) < 3)
    cmpSetterDispatch(STARTSUBASSIGN.OP, DFLTSUBASSIGN.OP,
      afun, place, call, cb, cntxt) ## punt
  else {
    nidx <- length(place) - 2
    if (nidx == 1)
      dflt.op <- list(code = VECSUBASSIGN.OP, rank = FALSE)
    else if (nidx == 2)
      dflt.op <- list(code = MATSUBASSIGN.OP, rank = FALSE)
    else
      dflt.op <- list(code = SUBASSIGN_N.OP, rank = TRUE)
    cmpSubassignDispatch(STARTSUBASSIGN.OP, dflt.op, afun, place, call,
      cb, cntxt)
  }
})
```
January 14, 2016

```
dflt.op <- list(code = SUBASSIGN2_N.OP, rank = TRUE)
cmpSubassignDispatch(STARTSUBASSIGN2_N.OP, dflt.op, afun, place, call, cb, cntxt)
}
```

Similarly, again, `cmpSubassignDispatch` is a variant of `cmpGetterDispatch` that passes index arguments on the stack.

```
cmpSubassignDispatch <- function(start.op, dflt.op, call, cb, cntxt) {
  if (dots.or.missing(call) || ! is.null(names(call)) || length(call) < 3)
    cntxt$stop(gettext("cannot compile this expression"), cntxt)
  else {
    ci <- cb$putconst(call)
    end.label <- cb$makelabel()
    cb$putcode(DUP2ND.OP)
    cb$putcode(start.op, ci, end.label)
    indices <- call[-c(1, 2)]
    for (i in seq_along(indices))
      cmp(indices[[i]], cb, cntxt, TRUE)
    if (dflt.op$rank)
      cb$putcode(dflt.op$code, ci, length(indices))
    else
      cb$putcode(dflt.op$code, ci)
    cb$putlabel(end.label)
    cb$putcode(SWAP.OP)
    TRUE
  }
}
```

And again the handlers fall back to `cmpGetterDispatch` if necessary.

```
setGetterInlineHandler("[", function(call, cb, cntxt) {
  if (dots.or.missing(call) || ! is.null(names(call)) || length(call) < 3)
    cmpGetterDispatch(STARTSUBSET.OP, DFLTSUBSET.OP, call, cb, cntxt)
  else {
    nidx <- length(call) - 2;
    if (nidx == 1)
      dflt.op <- list(code = VECSUBSET.OP, rank = FALSE)
    else if (nidx == 2)
      dflt.op <- list(code = MATSUBSET.OP, rank = FALSE)
    else
      dflt.op <- list(code = SUBSET_N.OP, rank = TRUE)
    cmpSubassignDispatch(STARTSUBSET_N.OP, dflt.op, call, cb, cntxt)
  }
})
```

```
setGetterInlineHandler("[[", function(call, cb, cntxt) {
```
```
if (dots.or.missing(call) || ! is.null(names(call)) || length(call) < 3)
  cmpGetterDispatch(STARTSUBSET2.OP, DFLTSUBSET2.OP, call, cb, cntxt)
else {
  nidx <- length(call) - 2;
  if (nidx == 1)
    dflt.op <- list(code = VECSUBSET2.OP, rank = FALSE)
  else if (nidx == 2)
    dflt.op <- list(code = MATSUBSET2.OP, rank = FALSE)
  else
    dflt.op <- list(code = SUBSET2_N.OP, rank = TRUE)
  cmpSubsetGetterDispatch(STARTSUBSET2_N.OP, dflt.op, call, cb, cntxt)
}

17 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 \texttt{pasteExpr} is used in the error messages.

\begin{align}
\texttt{pasteExpr} & \equiv \langle \texttt{pasteExpr} \texttt{ function} \rangle \\
\texttt{pasteExpr} & \leftarrow \texttt{function}(e, \texttt{prefix} = "\n ") \{ \\
\end{align}
The function `dots.or.missing` checks the argument list for any missing or ... arguments:

```r
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

```r
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.

```r
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.

```r
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.

$$\text{frameTypes function} \equiv$$

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

$$\text{find the number nl of local frames} \equiv$$

```r
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.

$$\text{find the number nn of namespace frames} \equiv$$

```r
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.

$$\text{find the number ng of global frames} \equiv$$

```r
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 namespace 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.

```r
defindHomeNS <- function(sym, ns) {
  if (exists(sym, ns, inherits = FALSE))
    ns
  else if (exists(".__NAMESPACE__.", ns, inherits = FALSE)) {
    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" if 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`.

```r
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.

```r
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⟩
} else {
  ⟨search in a selective import⟩
}

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.

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.

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.

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.

```r
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 utilities 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_PROFILE 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.

```r
bcprof <- function(expr) {
  .Internal(bcprofstart())
  expr
  .Internal(bcprofstop())
  val <- structure(.Internal(bcprofcounts()),
    names = 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.

```r
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

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>BCMISMATCH.OP</td>
<td>0</td>
</tr>
<tr>
<td>RETURN.OP</td>
<td>1</td>
</tr>
<tr>
<td>GOTO.OP</td>
<td>2</td>
</tr>
<tr>
<td>BRIFNOT.OP</td>
<td>3</td>
</tr>
<tr>
<td>POP.OP</td>
<td>4</td>
</tr>
<tr>
<td>DUP.OP</td>
<td>5</td>
</tr>
<tr>
<td>PRINTVALUE.OP</td>
<td>6</td>
</tr>
<tr>
<td>STARTLOOPCXTX.OP</td>
<td>7</td>
</tr>
<tr>
<td>ENDOOPCXTX.OP</td>
<td>8</td>
</tr>
<tr>
<td>DOOOPNEXT.OP</td>
<td>9</td>
</tr>
<tr>
<td>DOOOPBREAK.OP</td>
<td>10</td>
</tr>
<tr>
<td>STARTFOR.OP</td>
<td>11</td>
</tr>
<tr>
<td>STEPFOR.OP</td>
<td>12</td>
</tr>
<tr>
<td>ENDFOR.OP</td>
<td>13</td>
</tr>
<tr>
<td>SETLOOPVAL.OP</td>
<td>14</td>
</tr>
<tr>
<td>INVISIBLE.OP</td>
<td>15</td>
</tr>
<tr>
<td>LDCONST.OP</td>
<td>16</td>
</tr>
<tr>
<td>LDNULL.OP</td>
<td>17</td>
</tr>
<tr>
<td>LDFALSE.OP</td>
<td>18</td>
</tr>
<tr>
<td>LDTRUE.OP</td>
<td>19</td>
</tr>
<tr>
<td>GETVAR.OP</td>
<td>20</td>
</tr>
<tr>
<td>DDVAL.OP</td>
<td>21</td>
</tr>
<tr>
<td>SETVAR.OP</td>
<td>22</td>
</tr>
<tr>
<td>GETFUN.OP</td>
<td>23</td>
</tr>
<tr>
<td>GETGLOBFUN.OP</td>
<td>24</td>
</tr>
<tr>
<td>GETSYMFUN.OP</td>
<td>25</td>
</tr>
<tr>
<td>GETBUILTIN.OP</td>
<td>26</td>
</tr>
<tr>
<td>GETINTLBUILTIN.OP</td>
<td>27</td>
</tr>
<tr>
<td>CHECKFUN.OP</td>
<td>28</td>
</tr>
<tr>
<td>MAKEPROM.OP</td>
<td>29</td>
</tr>
<tr>
<td>DOMISSING.OP</td>
<td>30</td>
</tr>
<tr>
<td>SETTAG.OP</td>
<td>31</td>
</tr>
<tr>
<td>DODOTS.OP</td>
<td>32</td>
</tr>
<tr>
<td>PUSHARG.OP</td>
<td>33</td>
</tr>
<tr>
<td>PUSHCONSTARG.OP</td>
<td>34</td>
</tr>
<tr>
<td>PUSHNULLARG.OP</td>
<td>35</td>
</tr>
<tr>
<td>PUSHTREUARG.OP</td>
<td>36</td>
</tr>
<tr>
<td>PUSHPHASEARG.OP</td>
<td>37</td>
</tr>
<tr>
<td>CALL.OP</td>
<td>38</td>
</tr>
<tr>
<td>CALLBUILTIN.OP</td>
<td>39</td>
</tr>
<tr>
<td>CALLSPECIAL.OP</td>
<td>40</td>
</tr>
<tr>
<td>MAKECLOSURE.OP</td>
<td>41</td>
</tr>
<tr>
<td>UMINUS.OP</td>
<td>42</td>
</tr>
<tr>
<td>UPLUS.OP</td>
<td>43</td>
</tr>
</tbody>
</table>
ADD.OP <- 44
SUB.OP <- 45
MUL.OP <- 46
DIV.OP <- 47
EXPT.OP <- 48
SQRTP. 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
DOTSER. 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

\[ \text{Opcode argument counts} \equiv \begin{cases} \text{BCMISMATCH.OP = 0,} \\
\text{RETURN.OP = 0,} \\
\text{GOTO.OP = 1,} \\
\text{BRIFNOT.OP = 2,} \\
\text{POP.OP = 0,} \\
\text{DUP.OP = 0,} \\
\text{PRINTVALUE.OP = 0,} \\
\text{STARTLOOPCNTXT.OP = 1,} \\
\text{ENDLOOPCNTXT.OP = 0,} \\
\text{DOLOOPNEXT.OP = 0,} \\
\text{DOLOOPBREAK.OP = 0,} \\
\text{STARTFOR.OP = 3,} \end{cases} \]
STEPFOR.OP = 1,
ENDFOR.OP = 0,
SETLOOPVAL.OP = 0,
INVISIBLE.OP = 0,
LCONST.OP = 1,
LDNULL.OP = 0,
LDTTRUE.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,
DOTS.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
)

\langle opcode\ names \rangle \equiv
Opnames <- names(Opnames.argc)

E Implementation file

\langle cmp.R \rangle \equiv
# Automatically generated from ../noweb/compiler.nw.
#
# File src/library/compiler/R/cmp.R
# Part of the R package, https://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
# https://www.R-project.org/Licenses/

##
## Compiler options
##

\langle compiler\ options\ data\ base \rangle
## General Utilities

- `getCompilerOption function 18c`
- `pasteExpr function 91`
- `dots.or.missing function 92a`
- `any.dots function 92b`
- `is.ddsym function 92c`
- `missingArgs function 92d`

## Environment utilities

- `frameTypes function 93a`
- `findHomeNS function 94a`
- `packFrameName function 95c`
- `nsName function 96a`

## 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` [79b]
- `constNames definition` [79d]
- `checkConst function` [79a]
- `constantFoldSym function` [79c]
- `getFoldFun function` [80c]
- `constantFoldCall function` [80a]
- `constantFold function` [78b]
- `foldFuns definition` [80b]
- `languageFuns definition` [30a]

## Opcode constants
## Code buffer implementation

### make.codeBuf function

### codeBufCode function

### genCode function

## Compiler contexts

### make.toplevelContext function

### make.callContext function

### make.promiseContext function

### make.functionContext function

### make.nonTailCallContext function

### make.argContext function

### make.noValueContext function

### make.loopContext function

## Compiler top level

### cmp function

### cmpConst function
## Inlining mechanism

### Inline handler implementation

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.

### Inline handlers for some SPECIAL functions

inlining handler for left brace
inlining handler for if
inlining handler for &&
## Inline handlers for assignment expressions

- *setter inlining mechanism* 73b
- *getter inlining mechanism* 76b
- *cmpAssign function* 65b
- *flattenPlace function* 71b
- *cmpGetterCall function* 75c
- *checkAssign function* 66b
- *cmpSymbolAssign function* 67a
- *cmpComplexAssign function* 69b
- *cmpSetterCall function* 72a
- *getAssignFun function* 72c
- *cmpSetterDispatch function* 74b
- *inlining handlers for <-, =, and <<=* 66a
- *setter inline handler for $<-* 74a
- *setter inline handlers for [*< and [[<* 75a
- *cmpGetterDispatch function* 77b
- *getter inline handler for $* 77a
- *getter inline handlers for [ and [[* 78a

## 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 handler for log 51a⟩
⟨list of one argument math functions 51b⟩
⟨cmpMath1 function 51c⟩
⟨inline one argument math functions 52a⟩
⟨inline handlers for comparison operators 52b⟩
⟨inline handlers for & and | 52c⟩
## Inline handlers for the left parenthesis function

## Inline handlers for general BUILTIN and SPECIAL functions

## Inline handlers for subsetting and related operators

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

## Inline handlers for the family of is.xyz primitives
### Inlining handlers for is.xyz functions

## Default inline handlers for BUILTIN and SPECIAL functions

### install default inlining handlers

## Inline handlers for some .Internal functions

### simpleFormals function

### simpleArgs function

### is.simpleInternal function

### inlineSimpleInternalCall function

### cmpSimpleInternal function

### inline safe simple .Internal functions from base

### inline safe simple .Internal functions from stats

## Inline handler for switch

### findActionIndex function

### inline handler for switch

## Inline handler for .Call

### inline handler for .Call

## Inline handlers for generating integer sequences
## Inline handlers for integer sequences

## Inline handlers to control warnings

- `cmpMultiColon function` 60a
- `inlining handlers for :: and ::::` 60b
- `setter inlining handler for @<-` 75b
- `inlining handler for with` 60c
- `inlining handler for require` 61a

## Compiler warnings

- `suppressAll function` 18d
- `suppressUndef function` 18e
- `notifyLocalFun function` 19c
- `notifyUndefFun function` 19a
- `notifyUndefVar function` 19b
- `notifyNoSuperAssignVar function` 20c
- `notifyWrongArgCount function` 20b
- `notifyWrongDotsUse function` 20a
- `notifyWrongBreakNext function` 20d
- `notifyBadCall function` 20c
- `notifyBadAssignFun function` 20f
- `notifyMultipleSwitchDefaults function` 21a
## Compiler interface

**notifyNoSwitchcases** function

### Disassembler

**bcDecode** function

### Experimental Utilities

**bcprof** function
## Improved subset and subassign handling

`cmpSubsetDispatch function` (87b)

`inline handlers for subsetting` (88a)

`cmpSubassignDispatch function` (88b)

`inline handlers for subassignment` (89)

`cmpSubsetGetterDispatch function` (90a)

`inline handlers for subset getters` (90b)