NAME

Contextual::Return − Create context−sensitive return values

VERSION

This document describes Contextual::Return version 0.004014

SYNOPSIS

```
use Contextual::Return;
use Carp;

sub foo {
    return
        SCALAR { 'thirty-twelve' }
        LIST   { 1,2,3 }

        BOOL { 1 }
        NUM  { 7*6 }
        STR  { 'forty-two' }

        HASHREF  { {name => 'foo', value => 99} }
        ARRAYREF { [3,2,1] }

        GLOBREF  { \*STDOUT }
        CODEREF  { croak "Don't use this result as code!"; }
    ;
}

# and later...

if (my $foo = foo()) {
    for my $count (1..$foo) {
        print "$count: $foo is:\n"
            . "  array: @{$foo}\n"
            . "  hash:  $foo->{name} => $foo->{value}\n"
            ;
    }
    print {$foo} $foo->();
}
```

DESCRIPTION

Usually, when you need to create a subroutine that returns different values in different contexts (list, scalar, or void), you write something like:

```
sub get_server_status {
    my ($server_ID) = @_;

    # Acquire server data somehow...
    my %server_data = _ascertain_server_status($server_ID);

    # Return different components of that data,
    # depending on call context...
    if (wantarray()) {
        return @server_data{ qw(name uptime load users) };
    }
    if (defined wantarray()) {
        return $server_data{load};
    }
```

```
    if (!defined wantarray()) {
        carp 'Useless use of get_server_status() in void context';
        return;
    }
    else {
        croak q{Bad context! No biscuit!};
    }
}
```

That works okay, but the code could certainly be more readable. In its simplest usage, this module makes that code more readable by providing three subroutines — `LIST()`, `SCALAR()`, `VOID()` −−that are true only when the current subroutine is called in the corresponding context:

```
use Contextual::Return;

sub get_server_status {
    my ($server_ID) = @_;

    # Acquire server data somehow...
    my %server_data = _ascertain_server_status($server_ID);

    # Return different components of that data
    # depending on call context...
    if (LIST)   { return @server_data{ qw(name uptime load users) } }
    if (SCALAR) { return $server_data{load}                        }
    if (VOID)   { print "$server_data{load}\n"                     }
    else        { croak q{Bad context! No biscuit!}                }
}
```

Contextual returns

Those three subroutines can also be used in another way: as labels on a series of *contextual return blocks* (collectively known as a *contextual return sequence*). When a context sequence is returned, it automatically selects the appropriate contextual return block for the calling context. So the previous example could be written even more cleanly as:

```
use Contextual::Return;

sub get_server_status {
    my ($server_ID) = @_;

    # Acquire server data somehow...
    my %server_data = _ascertain_server_status($server_ID);

    # Return different components of that data
    # depending on call context...
    return (
        LIST    { return @server_data{ qw(name uptime load users) } }
        SCALAR  { return $server_data{load}                        }
        VOID    { print "$server_data{load}\n"                     }
        DEFAULT { croak q{Bad context! No biscuit!}                }
    );
}
```

The context sequence automatically selects the appropriate block for each call context.

Lazy contextual return values

`LIST` and `VOID` blocks are always executed during the `return` statement. However, scalar return blocks (`SCALAR`, `STR`, `NUM`, `BOOL`, etc.) blocks are not. Instead, returning any of scalar block types causes the

subroutine to return an object that lazily evaluates that block only when the return value is used.

This means that returning a `SCALAR` block is a convenient way to implement a subroutine with a lazy return value. For example:

```
sub digest {
    return SCALAR {
        my ($text) = @_;
        md5($text);
    }
}

my $digest = digest($text);

print $digest;   # md5() called only when $digest used as string
```

To better document this usage, the `SCALAR` block has a synonym: `LAZY`.

```
sub digest {
    return LAZY {
        my ($text) = @_;
        md5($text);
    }
}
```

Active contextual return values

Once a return value has been lazily evaluated in a given context, the resulting value is cached, and thereafter reused in that same context.

However, you can specify that, rather than being cached, the value should be re-evaluated *every* time the value is used:

```
 sub make_counter {
    my $counter = 0;
    return ACTIVE
        SCALAR   { ++$counter }
        ARRAYREF { [1..$counter] }
}

my $idx = make_counter();

print "$idx\n";       # 1
print "$idx\n";       # 2
print "[@$idx]\n";    # [1 2]
print "$idx\n";       # 3
print "[@$idx]\n";    # [1 2 3]
```

Semi-lazy contextual return values

Sometimes, single or repeated lazy evaluation of a scalar return value in different contexts isn't what you really want. Sometimes what you really want is for the return value to be lazily evaluated once only (the first time it's used in any context), and then for that first value to be reused whenever the return value is subsequently reevaluated in any other context.

To get that behaviour, you can use the `FIXED` modifier, which causes the return value to morph itself into the actual value the first time it is used. For example:

```
sub lazy {
    return
        SCALAR { 42 }
        ARRAYREF { [ 1, 2, 3 ] }
    ;
}

my $lazy = lazy();
print $lazy + 1;            # 43
print "@{$lazy}";           # 1 2 3


sub semilazy {
    return FIXED
        SCALAR { 42 }
        ARRAYREF { [ 1, 2, 3 ] }
    ;
}

my $semi = semilazy();
print $semi + 1;            # 43
print "@{$semi}";           # die q{Can't use string ("42") as an ARRAY ref}
```

Finer distinctions of scalar context

Because the scalar values returned from a context sequence are lazily evaluated, it becomes possible to be more specific about *what kind* of scalar value should be returned: a boolean, a number, or a string. To support those distinctions, Contextual::Return provides four extra context blocks: `NUM`, `STR`, `BOOL`, and `PUREBOOL`:

```
sub get_server_status {
    my ($server_ID) = @_;

    # Acquire server data somehow...
    my %server_data = _ascertain_server_status($server_ID);

    # Return different components of that data
    # depending on call context...
    return (
           LIST { @server_data{ qw(name uptime load users) } }
      PUREBOOL { $_ = $server_data{uptime}; $server_data{uptime} > 0 }
           BOOL { $server_data{uptime} > 0                   }
           NUM  { $server_data{load}                         }
           STR  { "$server_data{name}: $server_data{uptime}" }
           VOID { print "$server_data{load}\n"               }
        DEFAULT { croak q{Bad context! No biscuit!}          }
    );
}
```

With these in place, the object returned from a scalar-context call to `get_server_status()` now behaves differently, depending on how it's used. For example:

```
if ( my $status = get_server_status() ) {  # BOOL: True if uptime > 0
    $load_distribution[$status]++;         # INT:  Evaluates to load value
    print "$status\n";                     # STR:  Prints "name: uptime"
}

if (get_server_status()) {                 # PUREBOOL: also sets $_;
```

```
    print;                                  # ...which is then used here
}
```

Boolean vs Pure Boolean contexts

There is a special subset of boolean contexts where the return value is being used and immediately thrown away. For example, in the loop:

```
while (get_data()) {
    ...
}
```

the value returned by `get_data()` is tested for truth and then discarded. This is known as ''pure boolean context''. In contrast, in the loop:

```
while (my $data = get_data()) {
    ...
}
```

the value returned by `get_data()` is first assigned to `$data`, then tested for truth. Because of the assignment, the return value is *not* discarded after the boolean test. This is ordinary ''boolean context''.

In Perl, pure boolean context is often associated with a special side-effect, that does not occur in regular boolean contexts. For example:

```
while (<>) {...}          # $_ set as side-effect of pure boolean context

while ($v = <>) {...}     # $_ NOT set in ordinary boolean context
```

Contextual::Return supports this with a special subcase of `BOOL` named <PUREBOOL>. In pure boolean contexts, Contextual::Return will call a `PUREBOOL` handler if one has been defined, or fall back to a `BOOL` or `SCALAR` handler if no `PUREBOOL` handler exists. In ordinary boolean contexts only the `BOOL` or `SCALAR` handlers are tried, even if a `PUREBOOL` handler is also defined.

Typically `PUREBOOL` handlers are set up to have some side-effect (most commonly: setting `$_` or <$@>), like so:

```
sub get_data {
    my ($succeeded, @data) = _go_and_get_data();

    return
        PUREBOOL { $_ = $data[0]; $succeeded; }
            BOOL {                $succeeded; }
          SCALAR {                $data[0];   }
            LIST {                @data;      }
}
```

However, there is no requirement that they have side-effects. For example, they can also be used to implement ''look-but-don't-retrieve-yet'' checking:

```
sub get_data {
    my $data;
    return
        PUREBOOL {        _check_for_but_dont_get_data();   }
            BOOL { defined( $data ||= _go_and_get_data() ); }
             REF {          $data ||= _go_and_get_data();   }
}
```

Self-reference within handlers

Any handler can refer to the contextual return object it is part of, by calling the `RETOBJ()` function. This is particularly useful for `PUREBOOL` and `LIST` handlers. For example:

```
return
    PUREBOOL { $_ = RETOBJ; next handler; }
        BOOL { !$failed;                   }
     DEFAULT { $data;                      };
```

Referential contexts

The other major kind of scalar return value is a reference. Contextual::Return provides contextual return blocks that allow you to specify what to (lazily) return when the return value of a subroutine is used as a reference to a scalar (`SCALARREF {...}`), to an array (`ARRAYREF {...}`), to a hash (`HASHREF {...}`), to a subroutine (`CODEREF {...}`), or to a typeglob (`GLOBREF {...}`).

For example, the server status subroutine shown earlier could be extended to allow it to return a hash reference, thereby supporting "named return values":

```
sub get_server_status {
    my ($server_ID) = @_;

    # Acquire server data somehow...
    my %server_data = _ascertain_server_status($server_ID);

    # Return different components of that data
    # depending on call context...
    return (
           LIST { @server_data{ qw(name uptime load users) }  }
           BOOL { $server_data{uptime} > 0                    }
            NUM { $server_data{load}                          }
            STR { "$server_data{name}: $server_data{uptime}"  }
           VOID { print "$server_data{load}\n"                 }
        HASHREF { return \%server_data                        }
        DEFAULT { croak q{Bad context! No biscuit!}           }
    );
}

# and later...

my $users = get_server_status->{users};


# or, lazily...

my $server = get_server_status();

print "$server->{name} load = $server->{load}\n";
```

Interpolative referential contexts

The `SCALARREF {...}` and `ARRAYREF {...}` context blocks are especially useful when you need to interpolate a subroutine into strings. For example, if you have a subroutine like:

```
sub get_todo_tasks {
    return (
        SCALAR { scalar @todo_list }      # How many?
        LIST   { @todo_list        }      # What are they?
    );
}

# and later...
```

```
print "There are ", scalar(get_todo_tasks()), " tasks:\n",
        get_todo_tasks();
```

then you could make it much easier to interpolate calls to that subroutine by adding:

```
sub get_todo_tasks {
    return (
        SCALAR { scalar @todo_list }      # How many?
        LIST   { @todo_list        }      # What are they?

        SCALARREF { \scalar @todo_list }  # Ref to how many
        ARRAYREF  { \@todo_list        }  # Ref to them
    );
}

# and then...

print "There are ${get_todo_tasks()} tasks:\n@{get_todo_tasks()}";
```

In fact, this behaviour is so useful that it's the default. If you don't provide an explicit `SCALARREF {...}` block, Contextual::Return automatically provides an implicit one that simply returns a reference to whatever would have been returned in scalar context. Likewise, if no `ARRAYREF {...}` block is specified, the module supplies one that returns the list-context return value wrapped up in an array reference.

So you could just write:

```
sub get_todo_tasks {
    return (
        SCALAR { scalar @todo_list }      # How many?
        LIST   { @todo_list        }      # What are they?
    );
}

# and still do this...

print "There are ${get_todo_tasks()} tasks:\n@{get_todo_tasks()}";
```

Fallback contexts

As the previous sections imply, the `BOOL {...}`, `NUM {...}`, `STR {...}`, and various `*REF {...}` blocks, are special cases of the general `SCALAR {...}` context block. If a subroutine is called in one of these specialized contexts but does not use the corresponding context block, then the more general `SCALAR {...}` block is used instead (if it has been specified).

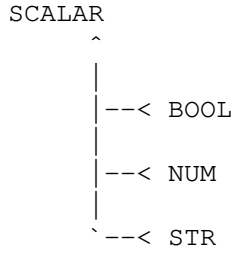
So, for example:

```
sub read_value_from {
    my ($fh) = @_;

    my $value = <$fh>;
    chomp $value;

    return (
        BOOL   { defined $value }
        SCALAR { $value         }
    );
}
```

ensures that the `read_value_from()` subroutine returns true in boolean contexts if the read was successful. But, because no specific `NUM {...}` or `STR {...}` return behaviours were specified, the

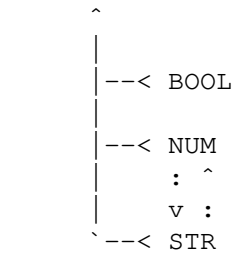
subroutine falls back on using its generic `SCALAR {...}` block in all other scalar contexts.

Another way to think about this behaviour is that the various kinds of scalar context blocks form a hierarchy:



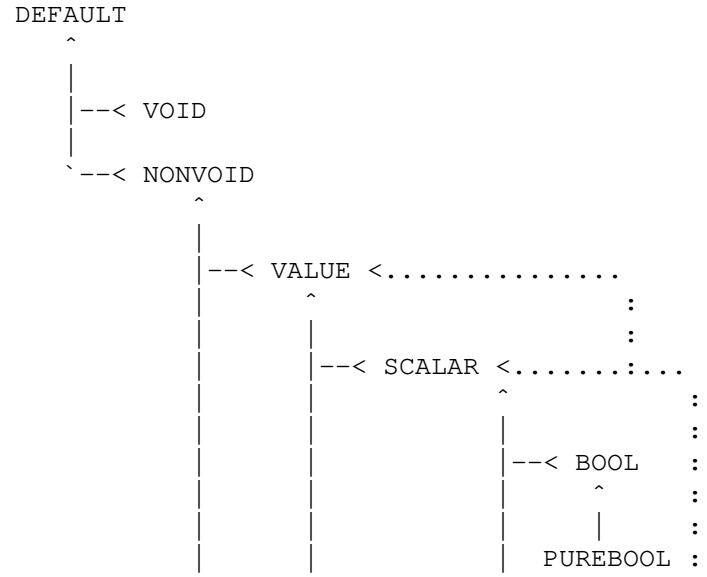
Contextual::Return uses this hierarchical relationship to choose the most specific context block available to handle any particular return context, working its way up the tree from the specific type it needs, to the more general type, if that's all that is available.

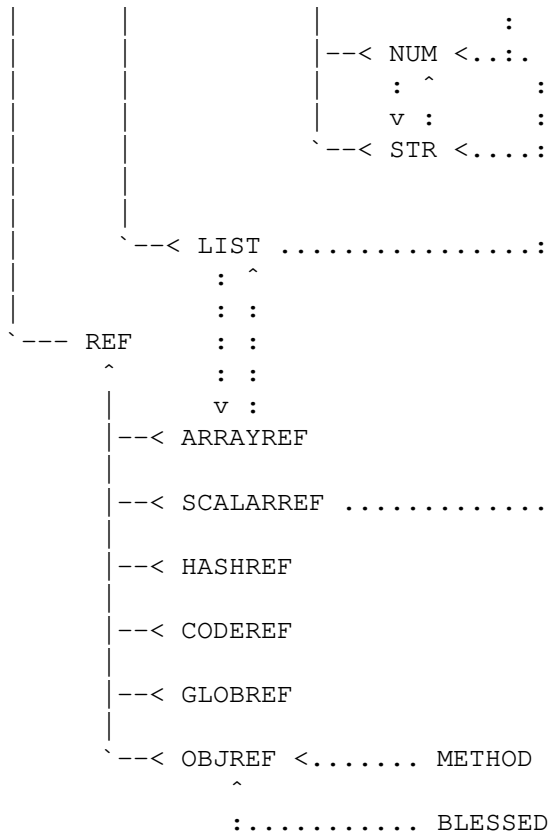
There are two slight complications to this picture. The first is that Perl treats strings and numbers as interconvertable so the diagram (and the Contextual::Return module) also has to allow these interconversions as a fallback strategy:



The dotted lines are meant to indicate that this intraconversion is secondary to the main hierarchical fallback. That is, in a numeric context, a `STR {...}` block will only be used if there is no `NUM {...}` block *and* no `SCALAR {...}` block. In other words, the generic context type is always used in preference to string<->number conversion.

The second slight complication is that the above diagram only shows a small part of the complete hierarchy of contexts supported by Contextual::Return. The full fallback hierarchy (including dotted interconversions) is:





As before, each dashed arrow represents a fallback relationship. That is, if the required context specifier isn't available, the arrows are followed until a more generic one is found. The dotted arrows again represent the interconversion of return values, which is attempted only after the normal hierarchical fallback fails.

For example, if a subroutine is called in a context that expects a scalar reference, but no `SCALARREF {...}` block is provided, then Contextual::Return tries the following blocks in order:

```
    REF {...}
NONVOID {...}
DEFAULT {...}
    STR {...} (automatically taking a reference to the result)
    NUM {...} (automatically taking a reference to the result)
 SCALAR {...} (automatically taking a reference to the result)
  VALUE {...} (automatically taking a reference to the result)
```

Likewise, in a list context, if there is no `LIST {...}` context block, the module tries:

```
   VALUE {...}
 NONVOID {...}
 DEFAULT {...}
ARRAYREF {...} (automatically dereferencing the result)
     STR {...} (treating it as a list of one element)
     NUM {...} (treating it as a list of one element)
  SCALAR {...} (treating it as a list of one element)
```

The more generic context blocks are especially useful for intercepting unexpected and undesirable call contexts. For example, to turn *off* the automatic scalar-ref and array-ref interpolative behaviour described in "Interpolative referential contexts", you could intercept *all* referential contexts using a generic `REF {...}` context block:

```
sub get_todo_tasks {
    return (
        SCALAR { scalar @todo_list }      # How many?
        LIST   { @todo_list        }      # What are they?

        REF { croak q{get_todo_task() can't be used as a reference} }
    );
}

print 'There are ', get_todo_tasks(), '...';    # Still okay
print "There are ${get_todo_tasks()}...";       # Throws an exception
```

Treating return values as objects

Normally, when a return value is treated as an object (i.e. has a method called on it), Contextual::Return invokes any `OBJREF` handler that was specified in the contextual return list, and delegates the method call to the object returned by that handler.

However, you can also be more specific, by specifying a `METHOD` context handler in the contextual return list. The block of this handler is expected to return one or more method–name/method–handler pairs, like so:

```
return
    METHOD {
        get_count => sub { my $n = shift; $data[$n]{count} },
        get_items => sub { my $n = shift; $data[$n]{items} },
        clear     => sub { @data = (); },
        reset     => sub { @data = (); },
    }
```

Then, whenever one of the specified methods is called on the return value, the corresponding subroutine will be called to implement it.

The method handlers must always be subroutine references, but the method-name specifiers may be strings (as in the previous example) or they may be specified generically, as either regexes or array references. Generic method names are used to call the same handler for two or more distinct method names. For example, the previous example could be simplified to:

```
return
    METHOD {
        qr/get_(\w+)/     => sub { my $n = shift; $data[$n]{$1} },
        ['clear','reset'] => sub { @data = (); },
    }
```

A method name specified by regex will invoke the corresponding handler for any method call request that the regex matches. A method name specified by array ref will invoke the corresponding handler if the method requested matches any of the elements of the array (which may themselves be strings or regexes).

When the method handler is invoked, the name of the method requested is passed to the handler in `$_`, and the method's argument list is passed (as usual) via `@_`.

Note that any methods not explicitly handled by the `METHOD` handlers will still be delegated to the object returned by the `OBJREF` handler (if it is also specified).

Not treating return values as objects

The use of `OBJREF` and `METHOD` are slightly complicated by the fact that contextual return values are themselves objects.

For example, prior to version 0.4.4 of the module, if you passed a contextual return value to `Scalar::Util::blessed()`, it always returned a true value (namely, the string: 'Contextual::Return::Value'), even if the return value had not specified handlers for `OBJREF` or `METHOD`.

In other words, the *implementation* of contextual return values (as objects) was getting in the way of the *use*

of contextual return values (as non-objects).

So the module now also provides a `BLESSED` handler, which allows you to explicitly control how contextual return values interact with `Scalar::Util::blessed()`.

If `$crv` is a contextual return value, by default `Scalar::Util::blessed($crv)` will now only return true if that return value has a `OBJREF`, `LAZY`, `REF`, `SCALAR`, `VALUE`, `NONVOID`, or `DEFAULT` handler that in turn returns a blessed object.

However if `$crv` also provides a `BLESSED` handler, `blessed()` will return whatever that handler returns.

This means:

```
sub simulate_non_object {
    return BOOL { 1 }
           NUM { 42 }
}

sub simulate_real_object {
    return OBJREF { bless {}, 'My::Class' }
            BOOL { 1 }
             NUM { 42 }
}

sub simulate_faked_object {
    return BLESSED { 'Foo' }
            BOOL { 1 }
             NUM { 42 }
}

sub simulate_previous_behaviour {
    return BLESSED { 'Contextual::Return::Value' }
            BOOL { 1 }
             NUM { 42 }
}


say blessed( simulate_non_object()          );   # undef
say blessed( simulate_real_object()         );   # My::Class
say blessed( simulate_faked_object()        );   # Foo
say blessed( simulate_previous_behaviour() );   # Contextual::Return::Value
```

Typically, you either want no `BLESSED` handler (in which case contextual return values pretend not to be blessed objects), or you want `BLESSED { 'Contextual::Return::Value' }` for backwards compatibility with pre−v0.4.7 behaviour.

Preventing fallbacks

Sometimes fallbacks can be too helpful. Or sometimes you want to impose strict type checking on a return value.

Contextual::Returns allows that via the `STRICT` specifier. If you include `STRICT` anywhere in your return statement, the module disables all fallbacks and will therefore through an exception if the return value is used in any way not explicitly specified in the contextual return sequence.

For example, to create a subroutine that returns only a string:

```
sub get_name {
    return STRICT STR { 'Bruce' }
}
```

If the return value of the subroutine is used in any other way than as a string, an exception will be thrown.

You can still specify handlers for more than a single kind of context when using `STRICT`:

```
sub get_name {
    return STRICT
        STR  { 'Bruce' }
        BOOL { 0 }
}
```

...but these will still be the only contexts in which the return value can be used:

```
my $n = get_name() ? 1 : 2;  # Okay because BOOL handler specified

my $n = 'Dr' . get_name();   # Okay because STR handler specified

my $n = 1 + get_name();      # Exception thrown because no NUM handler
```

In other words, `STRICT` allows you to impose strict type checking on your contextual return value.

Deferring handlers

Because the various handlers form a hierarchy, it's possible to implement more specific handlers by falling back on ("deferring to") more general ones. For example, a `PUREBOOL` handler is almost always identical in its basic behaviour to the corresponding `BOOL` handler, except that it adds some side-effect. For example:

```
return
    PUREBOOL { $_ = $return_val; defined $return_val && $return_val > 0 }
        BOOL {                   defined $return_val && $return_val > 0 }
      SCALAR {                   $return_val;                           }
```

So Contextual::Return allows you to have a handler perform some action and then defer to a more general handler to supply the actual return value. To fall back to a more general case in this way, you simply write:

```
next handler;
```

at the end of the handler in question, after which Contextual::Return will find the next-most-specific handler and execute it as well. So the previous example, could be re-written:

```
return
    PUREBOOL { $_ = $return_val; next handler;        }
        BOOL { defined $return_val && $return_val > 0 }
      SCALAR { $return_val;                           }
```

Note that *any* specific handler can defer to a more general one in this same way. For example, you could provide consistent and maintainable type-checking for a subroutine that returns references by providing `ARRAYREF`, `HASHREF`, and `SCALARREF` handlers that all defer to a generic `REF` handler, like so:

```
my $retval = _get_ref();

return
   SCALARREF { croak 'Type mismatch' if ref($retval) ne 'SCALAR';
               next handler;
             }
    ARRAYREF { croak 'Type mismatch' if ref($retval) ne 'ARRAY';
               next handler;
             }
     HASHREF { croak 'Type mismatch' if ref($retval) ne 'HASH';
               next handler;
```

```
            }
        REF { $retval }
```

If, at a later time, the process of returning a reference became more complex, only the `REF` handler would have to be updated.

Nested handlers

Another way of factoring out return behaviour is to nest more specific handlers inside more general ones. For instance, in the final example given in "Boolean vs Pure Boolean contexts":

```
sub get_data {
    my $data;
    return
        PUREBOOL {        _check_for_but_dont_get_data();    }
            BOOL { defined( $data ||= _go_and_get_data() ); }
             REF {          $data ||= _go_and_get_data();    }
}
```

you could factor out the repeated calls to `_go_and_get_data()` like so:

```
sub get_data {
    return
        PUREBOOL { _check_for_but_dont_get_data(); }
         DEFAULT {
            my $data = _go_and_get_data();

            BOOL { defined $data; }
             REF {         $data; }
         }
}
```

Here, the `DEFAULT` handler deals with every return context except pure boolean. Within that `DEFAULT` handler, the data is first retrieved, and then two "sub-handlers" deal with the ordinary boolean and referential contexts.

Typically nested handlers are used in precisely this way: to optimize for inexpensive special cases (such as pure boolean or integer or void return contexts) and only do extra work for those other cases that require it.

Failure contexts

Two of the most common ways to specify that a subroutine has failed are to return a false value, or to throw an exception. The Contextual::Return module provides a mechanism that allows the subroutine writer to support *both* of these mechanisms at the same time, by using the `FAIL` specifier.

A return statement of the form:

```
return FAIL;
```

causes the surrounding subroutine to return `undef` (i.e. false) in boolean contexts, and to throw an exception in any other context. For example:

```
use Contextual::Return;

sub get_next_val {
    my $next_val = <>;
    return FAIL if !defined $next_val;
    chomp $next_val;
    return $next_val;
}
```

If the `return FAIL` statement is executed, it will either return false in a boolean context:

```
if (my $val = get_next_val()) {      # returns undef if no next val
    print "[$val]\n";
}
```

or else throw an exception if the return value is used in any other context:

```
print get_next_val();       # throws exception if no next val

my $next_val = get_next_val();
print "[$next_val]\n";      # throws exception if no next val
```

The exception that is thrown is of the form:

```
Call to main::get_next_val() failed at demo.pl line 42
```

but you can change that message by providing a block to the `FAIL`, like so:

```
return FAIL { "No more data" } if !defined $next_val;
```

in which case, the final value of the block becomes the exception message:

```
No more data at demo.pl line 42
```

A failure value can be interrogated for its error message, by calling its `error()` method, like so:

```
my $val = get_next_val();
if ($val) {
    print "[$val]\n";
}
else {
    print $val->error, "\n";
}
```

Configurable failure contexts

The default `FAIL` behaviour — false in boolean context, fatal in all others — works well in most situations, but violates the Platinum Rule ("Do unto others as *they* would have done unto them").

So it may be user-friendlier if the user of a module is allowed decide how the module's subroutines should behave on failure. For example, one user might prefer that failing subs always return undef; another might prefer that they always throw an exception; a third might prefer that they always log the problem and return a special Failure object; whilst a fourth user might want to get back `0` in scalar contexts, an empty list in list contexts, and an exception everywhere else.

You could create a module that allows the user to specify all these alternatives, like so:

```
package MyModule;
use Contextual::Return;
use Log::StdLog;

sub import {
    my ($package, @args) = @_;

    Contextual::Return::FAIL_WITH {
        ':false' => sub { return undef },
        ':fatal' => sub { croak @_       },
        ':filed' => sub {
                        print STDLOG 'Sub ', (caller 1)[3], ' failed';
                        return Failure->new();
                    },
        ':fussy' => sub {
                        SCALAR  { undef      }
                        LIST    { ()         }
                        DEFAULT { croak @_ }
```

```
                        },
    }, @args;
}
```

This configures Contextual::Return so that, instead of the usual false-or-fatal semantics, every `return FAIL` within MyModule's namespace is implemented by one of the four subroutines specified in the hash that was passed to `FAIL_WITH`.

Which of those four subs implements the `FAIL` is determined by the arguments passed after the hash (i.e. by the contents of `@args`). `FAIL_WITH` walks through that list of arguments and compares them against the keys of the hash. If a key matches an argument, the corresponding value is used as the implementation of `FAIL`. Note that, if subsequent arguments also match a key, their subroutine overrides the previously installed implementation, so only the final override has any effect. Contextual::Return generates warnings when multiple overrides are specified.

All of which mean that, if a user loaded the MyModule module like this:

```
use MyModule qw( :fatal other args here );
```

then every `FAIL` within MyModule would be reconfigured to throw an exception in all circumstances, since the presence of the `':fatal'` in the argument list will cause `FAIL_WITH` to select the hash entry whose key is `':fatal'`.

On the other hand, if they loaded the module:

```
use MyModule qw( :fussy other args here );
```

then each `FAIL` within MyModule would return undef or empty list or throw an exception, depending on context, since that's what the subroutine whose key is `':fussy'` does.

Many people prefer module interfaces with a *`flag => value`* format, and `FAIL_WITH` supports this too. For example, if you wanted your module to take a `-fail` flag, whose associated value could be any of `"undefined"`, `"exception"`, `"logged"`, or `"context"`, then you could implement that simply by specifying the flag as the first argument (i.e. *before* the hash) like so:

```
sub import {
    my $package = shift;

    Contextual::Return::FAIL_WITH -fail => {
        'undefined' => sub { return undef },
        'exception' => sub { croak @_ },
        'logged'    => sub {
                           print STDLOG 'Sub ', (caller 1)[3], ' failed';
                           return Failure->new();
                       },
        'context' => sub {
                           SCALAR  { undef      }
                           LIST    { ()         }
                           DEFAULT { croak @_ }
                       },
    }, @_;
```

and then load the module:

```
use MyModule qw( other args here ), -fail=>'undefined';
```

or:

```
use MyModule qw( other args here ), -fail=>'exception';
```

In this case, `FAIL_WITH` scans the argument list for a pair of values: its flag string, followed by some other selector value. Then it looks up the selector value in the hash, and installs the corresponding subroutine as its local `FAIL` handler.

If this ‘‘flagged’’ interface is used, the user of the module can also specify their own handler directly, by passing a subroutine reference as the selector value instead of a string:

```
use MyModule qw( other args here ), -fail=>sub{ die 'horribly'};
```

If this last example were used, any call to `FAIL` within MyModule would invoke the specified anonymous subroutine (and hence throw a ’horribly’ exception).

Note that, any overriding of a `FAIL` handler is specific to the namespace and file from which the subroutine that calls `FAIL_WITH` is itself called. Since `FAIL_WITH` is designed to be called from within a module’s `import()` subroutine, that generally means that the `FAIL`s within a given module X are only overridden for the current namespace within the particular file from module X is loaded. This means that two separate pieces of code (in separate files or separate namespaces) can each independently override a module’s `FAIL` behaviour, without interfering with each other.

Lvalue contexts

Recent versions of Perl offer (limited) support for lvalue subroutines: subroutines that return a modifiable variable, rather than a simple constant value.

Contextual::Return can make it easier to create such subroutines, within the limitations imposed by Perl itself. The limitations that Perl places on lvalue subs are:

1. The subroutine must be declared with an `:lvalue` attribute:

   ```
   sub foo :lvalue {...}
   ```

2. The subroutine must not return via an explicit `return`. Instead, the last statement must evaluate to a variable, or must be a call to another lvalue subroutine call.

   ```
   my ($foo, $baz);

   sub foo :lvalue {
       $foo;               # last statement evals to a var
   }

   sub bar :lvalue {
       foo();              # last statement is lvalue sub call
   }

   sub baz :lvalue {
       my ($arg) = @_;

       $arg > 0            # last statement evals...
           ? $baz          # ...to a var
           : bar();        # ...or to an lvalue sub call
   }
   ```

Thereafter, any call to the lvalue subroutine produces a result that can be assigned to:

```
baz(0) = 42;            # same as: $baz = 42

baz(1) = 84;            # same as:                  bar() = 84
                        #  which is the same as:    foo() = 84
                        #   which is the same as:   $foo  = 84
```

Ultimately, every lvalue subroutine must return a scalar variable, which is then used as the lvalue of the assignment (or whatever other lvalue operation is applied to the subroutine call). Unfortunately, because the subroutine has to return this variable *before* the assignment can take place, there is no way that a normal lvalue subroutine can get access to the value that will eventually be assigned to its return value.

This is occasionally annoying, so the Contextual::Return module offers a solution: in addition to all the context blocks described above, it provides three special contextual return blocks specifically for use in

lvalue subroutines: `LVALUE`, `RVALUE`, and `NVALUE`.

Using these blocks you can specify what happens when an lvalue subroutine is used in lvalue and non-lvalue (rvalue) context. For example:

```
my $verbosity_level = 1;

# Verbosity values must be between 0 and 5...
sub verbosity :lvalue {
    LVALUE { $verbosity_level = max(0, min($_, 5)) }
    RVALUE { $verbosity_level                      }
}
```

The `LVALUE` block is executed whenever `verbosity` is called as an lvalue:

```
verbosity() = 7;
```

The block has access to the value being assigned, which is passed to it as `$_`. So, in the above example, the assigned value of 7 would be aliased to `$_` within the `LVALUE` block, would be reduced to 5 by the "min-of-max" expression, and then assigned to `$verbosity_level`.

(If you need to access the caller's `$_`, it's also still available: as `$CALLER::_`.)

When the subroutine isn't used as an lvalue:

```
print verbosity();
```

the `RVALUE` block is executed instead and its final value returned. Within an `RVALUE` block you can use any of the other features of Contextual::Return. For example:

```
sub verbosity :lvalue {
    LVALUE { $verbosity_level = int max(0, min($_, 5)) }
    RVALUE {
        NUM  { $verbosity_level               }
        STR  { $description[$verbosity_level] }
        BOOL { $verbosity_level > 2           }
    }
}
```

but the context sequence must be nested inside an `RVALUE` block.

You can also specify what an lvalue subroutine should do when it is used neither as an lvalue nor as an rvalue (i.e. in void context), by using an `NVALUE` block:

```
sub verbosity :lvalue {
    my ($level) = @_;

    NVALUE { $verbosity_level = int max(0, min($level, 5)) }
    LVALUE { $verbosity_level = int max(0, min($_,     5)) }
    RVALUE {
        NUM  { $verbosity_level               }
        STR  { $description[$verbosity_level] }
        BOOL { $verbosity_level > 2           }
    }
}
```

In this example, a call to `verbosity()` in void context sets the verbosity level to whatever argument is passed to the subroutine:

```
verbosity(1);
```

Note that you *cannot* get the same effect by nesting a `VOID` block within an `RVALUE` block:

```
    LVALUE { $verbosity_level = int max(0, min($_, 5)) }
    RVALUE {
        NUM  { $verbosity_level               }
        STR  { $description[$verbosity_level] }
        BOOL { $verbosity_level > 2           }
        VOID { $verbosity_level = $level      }  # Wrong!
    }
```

That's because, in a void context the return value is never evaluated, so it is never treated as an rvalue, which means the `RVALUE` block never executes.

Result blocks

Occasionally, it's convenient to calculate a return value *before* the end of a contextual return block. For example, you may need to clean up external resources involved in the calculation after it's complete. Typically, this requirement produces a slightly awkward code sequence like this:

```
return
    VALUE {
        $db->start_work();
        my $result = $db->retrieve_query($query);
        $db->commit();
        $result;
    }
```

Such code sequences become considerably more awkward when you want the return value to be context sensitive, in which case you have to write either:

```
return
    LIST {
        $db->start_work();
        my @result = $db->retrieve_query($query);
        $db->commit();
        @result;
    }
    SCALAR {
        $db->start_work();
        my $result = $db->retrieve_query($query);
        $db->commit();
        $result;
    }
```

or, worse:

```
return
    VALUE {
        $db->start_work();
        my $result = LIST ? [$db->retrieve_query($query)]
                          :  $db->retrieve_query($query);
        $db->commit();
        LIST ? @{$result} : $result;
    }
```

To avoid these infelicities, Contextual::Return provides a second way of setting the result of a context block; a way that doesn't require that the result be the last statement in the block:

```
return
    LIST {
        $db->start_work();
        RESULT { $db->retrieve_query($query) };
        $db->commit();
    }
    SCALAR {
        $db->start_work();
        RESULT { $db->retrieve_query($query) };
        $db->commit();
    }
```

The presence of a `RESULT` block inside a contextual return block causes that block to return the value of the final statement of the `RESULT` block as the handler's return value, rather than returning the value of the handler's own final statement. In other words, the presence of a `RESULT` block overrides the normal return value of a context handler.

Better still, the `RESULT` block always evaluates its final statement in the same context as the surrounding `return`, so you can just write:

```
return
    VALUE {
        $db->start_work();
        RESULT { $db->retrieve_query($query) };
        $db->commit();
    }
```

and the `retrieve_query()` method will be called in the appropriate context in all cases.

A `RESULT` block can appear anywhere inside any contextual return block, but may not be used outside a context block. That is, this is an error:

```
if ($db->closed) {
    RESULT { undef }; # Error: not in a context block
}
return
    VALUE {
        $db->start_work();
        RESULT { $db->retrieve_query($query) };
        $db->commit();
    }
```

Post-handler clean-up

If a subroutine uses an external resource, it's often necessary to close or clean-up that resource after the subroutine ends...regardless of whether the subroutine exits normally or via an exception.

Typically, this is done by encapsulating the resource in a lexically scoped object whose destructor does the clean-up. However, if the clean-up doesn't involve deallocation of an object (as in the `$db->commit()` example in the previous section), it can be annoying to have to create a class and allocate a container object, merely to mediate the clean-up.

To make it easier to manage such resources, Contextual::Return supplies a special labelled block: the `RECOVER` block. If a `RECOVER` block is specified as part of a contextual return sequence, that block is executed after any context handler, even if the context handler exits via an exception.

So, for example, you could implement a simple commit-or-revert policy like so:

```
return
    LIST    { $db->retrieve_all($query)  }
    SCALAR  { $db->retrieve_next($query) }
    RECOVER {
        if ($@) {
            $db->revert();
        }
        else {
            $db->commit();
        }
    }
```

The presence of a `RECOVER` block also intercepts all exceptions thrown in any other context block in the same contextual return sequence. Any such exception is passed into the `RECOVER` block in the usual manner: via the `$@` variable. The exception may be rethrown out of the `RECOVER` block by calling `die`:

```
return
    LIST    { $db->retrieve_all($query) }
    DEFAULT { croak "Invalid call (not in list context)" }
    RECOVER {
        die $@ if $@;    # Propagate any exception
        $db->commit();   # Otherwise commit the changes
    }
```

A `RECOVER` block can also access or replace the returned value, by invoking a `RESULT` block. For example:

```
return
    LIST    { attempt_to_generate_list_for(@_)  }
    SCALAR  { attempt_to_generate_count_for(@_) }
    RECOVER {
        if ($@) {                # On any exception...
            warn "Replacing return value. Previously: ", RESULT;
            RESULT { undef }     # ...return undef
        }
    }
```

Post-return clean-up

Occasionally it's necessary to defer the clean-up of resources until after the return value has been used. Once again, this is usually done by returning an object with a suitable destructor.

Using Contextual::Return you can get the same effect, by providing a `CLEANUP` block in the contextual return sequence:

```
return
    LIST    { $db->retrieve_all($query)  }
    SCALAR  { $db->retrieve_next($query) }
    CLEANUP { $db->commit()              }
```

In this example, the `commit` method call is only performed after the return value has been used by the caller. Note that this is quite different from using a `RECOVER` block, which is called as the subroutine returns its value; a `CLEANUP` is called when the returned value is garbage collected.

A `CLEANUP` block is useful for controlling resources allocated to support an `ACTIVE` return value. For example:

```
my %file;

# Return an active value that is always the next line from a file...
sub readline_from {
```

```
        my ($file_name) = @_;

        # Open the file, if not already open...
        if (!$file{$file_name}) {
            open $file{$file_name}{handle}, '<', $file_name;
        }

        # Track how many active return values are using this file...
        $file{$file_name}{count}++;

        return ACTIVE
            # Evaluating the return value returns the next line...
            VALUE   { readline $file{$file_name}{handle} }

            # Once the active value is finished with, clean up the filehandle...
            CLEANUP {
                delete $file{$file_name}
                    if --$file{$file_name}{count} == 0;
            }
    }
```

Debugging contextual return values

Contextual return values are implemented as opaque objects (using the "inside-out" technique). This means that passing such values to Data::Dumper produces an uninformative output like:

```
$VAR1 = bless( do{\(my $o = undef)}, 'Contextual::Return::Value' );
```

So the module provides two methods that allow contextual return values to be correctly reported: either directly, or when dumped by Data::Dumper.

To dump a contextual return value directly, call the module's `DUMP()` method explicitly and print the result:

```
print $crv->Contextual::Return::DUMP();
```

This produces an output something like:

```
[
 { FROM       => 'main::foo'                                          },
 { NO_HANDLER => [ 'VOID', 'CODEREF', 'HASHREF', 'GLOBREF' ]          },
 { FALLBACKS  => [ 'VALUE' ]                                          },
 { LIST       => [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ]                   },
 { STR        => '<<<Throws exception: Died at demo.pl line 7.>>>'   },
 { NUM        => 42                                                   },
 { BOOL       => -1                                                   },
 { SCALARREF  => '<<<self-reference>>>'                               },
 { ARRAYREF   => [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ]                    },
];
```

The `FROM` hash entry names the subroutine that produced the return value. The `NO_HANDLER` hash entry lists those contexts for which no handler was defined (and which would therefore normally produce "can't call" exceptions such as: `"Can't call main::foo in VOID context"`). The `FALLBACKS` hash entry lists any "generic" contexts such as `VALUE`, `NONVOID`, `REF`, `DEFAULT`, etc. that the contextual return value can also handle. After these, all the remaining hash entries are actual contexts in which the return value could successfully be evaluated, and the value it would produce in each of those contexts.

The Data::Dumper module also has a mechanism by which you can tell it how to produce a similar listing automatically whenever a contextual return value is passed to its `Dumper` method. Data::Dumper allows you to register a "freezer" method, that is called prior to dumping, and which can be used to adapt an

opaque object to make it dumpable. Contextual::Return provides just such a method (`Contextual::Return::FREEZE()`) for you to register, like so:

```
use Data::Dumper 'Dumper';

local $Data::Dumper::Freezer = 'Contextual::Return::FREEZE';

print Dumper $foo;
```

The output is then precisely the same as `Contextual::Return::DUMP()` would produce.

Note that, with both of the above dumping mechanisms, it is essential to use the full name of the method. That is:

```
print $crv->Contextual::Return::DUMP();
```

rather than:

```
print $crv->DUMP();
```

This is because the shorter version is interpreted as calling the `DUMP()` method on the object returned by the return value's `OBJREF` context block (see "Scalar reference contexts")

For the same reason, you must write:

```
local $Data::Dumper::Freezer = 'Contextual::Return::FREEZE';
```

not:

```
local $Data::Dumper::Freezer = 'FREEZE';
```

Namespace controls

By default the module exports a large number of return context markers:

```
DEFAULT     REF          LAZY
VOID        SCALARREF    FIXED
NONVOID     ARRAYREF     ACTIVE
LIST        CODEREF      RESULT
SCALAR      HASHREF      RECOVER
VALUE       GLOBREF      CLEANUP
STR         OBJREF       RVALUE
NUM         METHOD       LVALUE
BOOL                     NVALUE
PUREBOOL
```

These are exported as subroutines, and so can conflict with existing subroutines in your namespace, or with subroutines imported from other modules.

Contextual::Return allows you to control which contextual return blocks are exported into any namespace that uses the module. It also allows you to rename blocks to avoid namespace conflicts with existing subroutines.

Both these features are controlled by passing arguments to the `use` statement that loads the module as follows:

- Any string passed as an argument to `use Contextual::Return`, exports only the block name it specifies;
- Any regex passed as an argument to `use Contextual::Return` exports every block name it matches;
- Any array ref (recursively) exports each of its elements
- Any string that appears immediately after one of the above three specifiers, and which is not itself a block name, renames the handlers exported by that preceding specifier by filtering each handler name through `sprintf()`

That is, you can specify handlers to be exported by exact name (as a string), by general pattern (as a regex), or collectively (in an array). And after any of these export specifications, you can append a template in which any `'%s'` will be replaced by the original name of the handler. For example:

```
# Selectively export specific sets of handlers...
use Contextual::Return  qr/[NLR]VALUE/;
use Contextual::Return  qr/.*REF/;

# Selective export specific sets and add a suffix to each...
use Contextual::Return  qr/[NLR]VALUE/ => '%s_CONTEXT';

# Selective export specific sets and add a prefix to each...
use Contextual::Return  qr/.*REF/ => 'CR_%s';

# Export a list of handlers...
use Contextual::Return    'NUM', 'STR', 'BOOL' ;
use Contextual::Return qw< NUM    STR    BOOL >;
use Contextual::Return   ['NUM', 'STR', 'BOOL'];

# Export a list of handlers, renaming them individually...
use Contextual::Return  NUM => 'NUMERIC', STR => 'TEXT', BOOL => 'CR_%s';

# Export a list of handlers, renaming them collectively...
use Contextual::Return  ['NUM', 'STR', 'BOOL'] => '%s_CONTEXT';

# Mixed exports and renames...
use Contextual::Return (
    STR => 'TEXT',
    ['NUM', 'BOOL'] => 'CR_%s',
    ['LIST', 'SCALAR', 'VOID', qr/^[NLR]VALUE/] => '%s_CONTEXT',
);
```

INTERFACE

Context tests

`LIST()`
: Returns true if the current subroutine was called in list context. A cleaner way of writing: `wantarray()`

`SCALAR()`
: Returns true if the current subroutine was called in scalar context. A cleaner way of writing: `defined wantarray() && ! wantarray()`

`VOID()`
: Returns true if the current subroutine was called in void context. A cleaner way of writing: `!defined wantarray()`

`NONVOID()`
: Returns true if the current subroutine was called in list or scalar context. A cleaner way of writing: `defined wantarray()`

Standard contexts

`LIST {...}`
: The block specifies what the context sequence should evaluate to when called in list context.

`SCALAR {...}`
: The block specifies what the context sequence should evaluate to in scalar contexts, unless some more-specific specifier scalar context specifier (see below) also occurs in the same context sequence.

`VOID {...}`
The block specifies what the context sequence should do when called in void context.

Scalar value contexts

`BOOL {...}`
The block specifies what the context sequence should evaluate to when treated as a boolean value.

`NUM {...}`
The block specifies what the context sequence should evaluate to when treated as a numeric value.

`STR {...}`
The block specifies what the context sequence should evaluate to when treated as a string value.

`LAZY {...}`
Another name for `SCALAR {...}`. Usefully self-documenting when the primary purpose of the contextual return is to defer evaluation of the return value until it's actually required.

Scalar reference contexts

`SCALARREF {...}`
The block specifies what the context sequence should evaluate to when treated as a reference to a scalar.

`ARRAYREF {...}`
The block specifies what the context sequence should evaluate to when treated as a reference to an array.

`HASHREF {...}`
The block specifies what the context sequence should evaluate to when treated as a reference to a hash.

Note that a common error here is to write:

HASHREF { a=>1, b=>2, c=>3 }

The curly braces there are a block, not a hash constructor, so the block doesn't return a hash reference and the interpreter throws an exception. What's needed is:

HASHREF { {a=>1, b=>2, c=>3} }

in which the inner braces *are* a hash constructor.

`CODEREF {...}`
The block specifies what the context sequence should evaluate to when treated as a reference to a subroutine.

`GLOBREF {...}`
The block specifies what the context sequence should evaluate to when treated as a reference to a typeglob.

`OBJREF {...}`
The block specifies what the context sequence should evaluate to when treated as a reference to an object.

`METHOD {...}`
The block can be used to specify particular handlers for specific method calls when the return value is treated as an object reference. It should return a list of methodname/methodbody pairs. Each method name can be specified as a string, a regex, or an array of strings or regexes. The method bodies must be specified as subroutine references (usually anonymous subs). The first method name that matches the actual method call selects the corresponding handler, which is then called.

Generic contexts

`VALUE {...}`
The block specifies what the context sequence should evaluate to when treated as a non-referential value (as a boolean, numeric, string, scalar, or list). Only used if there is no more-specific value context specifier in the context sequence.

`REF {...}`
The block specifies what the context sequence should evaluate to when treated as a reference of any kind. Only used if there is no more-specific referential context specifier in the context sequence.

`NONVOID {...}`
The block specifies what the context sequence should evaluate to when used in a non-void context of any kind. Only used if there is no more-specific context specifier in the context sequence.

`DEFAULT {...}`
The block specifies what the context sequence should evaluate to when used in a void or non-void context of any kind. Only used if there is no more-specific context specifier in the context sequence.

Failure context

`FAIL`
This block is executed unconditionally and is used to indicate failure. In a Boolean context it return false. In all other contexts it throws an exception consisting of the final evaluated value of the block.

That is, using `FAIL`:

return FAIL { ''Could not defenestrate the widget'' }

is exactly equivalent to writing:

return BOOL { 0 } DEFAULT { croak ''Could not defenestrate the widget'' }

except that the reporting of errors is a little smarter under `FAIL`.

If `FAIL` is called without specifying a block:

return FAIL;

it is equivalent to:

return FAIL { croak ''Call to <subname> failed'' }

(where `<subname>` is replaced with the name of the surrounding subroutine).

Note that, because `FAIL` implicitly covers every possible return context, it cannot be chained with other context specifiers.

`Contextual::Return::FAIL_WITH`
This subroutine is not exported, but may be called directly to reconfigure `FAIL` behaviour in the caller's namespace.

The subroutine is called with an optional string (the *flag*), followed by a mandatory hash reference (the *configurations hash*), followed by a list of zero-or-more strings (the *selector list*). The values of the configurations hash must all be subroutine references.

If the optional flag is specified, `FAIL_WITH` searches the selector list looking for that string, then uses the *following* item in the selector list as its *selector value*. If that selector value is a string, `FAIL_WITH` looks up that key in the hash, and installs the corresponding subroutine as the namespace's `FAIL` handler (an exception is thrown if the selector string is not a valid key of the configurations hash). If the selector value is a subroutine reference, `FAIL_WITH` installs that subroutine as the `FAIL` handler.

If the optional flag is *not* specified, `FAIL_WITH` searches the entire selector list looking for the last element that matches any key in the configurations hash. It then looks up that key in the hash, and installs the corresponding subroutine as the namespace's `FAIL` handler.

See ''Configurable failure contexts'' for examples of using this feature.

Lvalue contexts

`LVALUE`
This block is executed when the result of an `:lvalue` subroutine is assigned to. The assigned value is passed to the block as `$_`. To access the caller's `$_` value, use `$CALLER::_`.

`RVALUE`
: This block is executed when the result of an `:lvalue` subroutine is used as an rvalue. The final value that is evaluated in the block becomes the rvalue.

`NVALUE`
: This block is executed when an `:lvalue` subroutine is evaluated in void context.

Explicit result blocks

`RESULT`
: This block may only appear inside a context handler block. It causes the surrounding handler to return the final value of the `RESULT`'s block, rather than the final value of the handler's own block. This override occurs regardless of the location to the `RESULT` block within the handler.

  If called without a trailing `{...}`, it simply returns the current result value in scalar contexts, or the list of result values in list context.

Recovery blocks

`RECOVER`
: If present in a context return sequence, this block grabs control after any context handler returns or exits via an exception. If an exception was thrown it is passed to the `RECOVER` block via the `$@` variable.

Clean-up blocks

`CLEANUP`
: If present in a context return sequence, this block grabs control when a return value is garbage collected.

Modifiers

`FIXED`
: This specifies that the scalar value will only be evaluated once, the first time it is used, and that the value will then morph into that evaluated value.

`ACTIVE`
: This specifies that the scalar value's originating block will be re− evaluated every time the return value is used.

Debugging support

`$crv->Contextual::Return::DUMP()`
: Return a dumpable representation of the return value in all viable contexts.

```
local $Data::Dumper::Freezer = 'Contextual::Return::FREEZE';
local $Data::Dumper::Freezer = \&Contextual::Return::FREEZE;
```
: Configure Data::Dumper to correctly dump a representation of the contextual return value.

DIAGNOSTICS

`Can't use %s as export specifier`
: In your `use Contextual::Return` statement you specified something (such as a hash or coderef) that can't be used to select what the module exports. Make sure the list of selectors includes only strings, regexes, or references to arrays of strings or regexes.

`use Contextual::Return qr{%s} didn't export anything`
: In your `use Contextual::Return` statement you specified a regex to select which handlers to support, but the regex didn't select any handlers. Check that the regex you're using actually does match at least one of the names of the modules many handlers.

`Can't export %s: no such handler`
: In your `use Contextual::Return` statement you specified a string as the name of a context handler to be exported, but the module doesn't export a handler of that name. Check the spelling for the requested export.

`Can't call %s in a %s context`
`Can't use return value of %s in a %s context`
: The subroutine you called uses a contextual return, but doesn't specify what to return in the particular context in which you called it. You either need to change the context in which you're calling the subroutine, or else add a context block corresponding to the offending context (or perhaps a `DEFAULT {...}` block).

`Can't call bare %s {...} in %s context`
: You specified a handler (such as `VOID {...}` or `LIST {...}`) outside any subroutine, and in a context that it can't handle. Did you mean to place the handler outside of a subroutine? If so, then you need to put it in a context it can actually handle. Otherwise, perhaps you need to replace the trailing block with parens (that is: `VOID()` or `LIST()`).

`Call to %s at %s didn't return a %s reference"`
: You called the subroutine in a context that expected to get back a reference of some kind but the subroutine didn't specify the corresponding `SCALARREF`, `ARRAYREF`, `HASHREF`, `CODEREF`, `GLOBREF`, or generic `REF`, `NONVOID`, or `DEFAULT` handlers. You need to specify the appropriate one of these handlers in the subroutine.

`Can't call method '%s' on %s value returned by %s"`
: You called the subroutine and then tried to call a method on the return value, but the subroutine returned a classname or object that doesn't have that method. This probably means that the subroutine didn't return the classname or object you expected. Or perhaps you need to specify an `OBJREF {...}` context block.

`Can't install two %s handlers`
: You attempted to specify two context blocks of the same name in the same return context, which is ambiguous. For example:

```
sub foo: lvalue {
    LVALUE { $foo = $_ }
    RVALUE { $foo }
    LVALUE { $foo = substr($_,1,10) }
}
```

or:

```
sub bar {
    return
        BOOL { 0 }
        NUM  { 1 }
        STR  { "two" }
        BOOL { 1 };
}
```

Did you cut-and-paste wrongly, or mislabel one of the blocks?

`Expected a %s block after the %s block but found instead: %s`
: If you specify any of `LVALUE`, `RVALUE`, or `NVALUE`, then you can only specify `LVALUE`, `RVALUE`, or `NVALUE` blocks in the same return context. If you need to specify other contexts (like `BOOL`, or `STR`, or `REF`, etc.), put them inside an `RVALUE` block. See "Lvalue contexts" for an example.

`Call to %s failed at %s`
: This is the default exception that a `FAIL` throws in a non-scalar context. Which means that the subroutine you called has signalled failure by throwing an exception, and you didn't catch that exception. You should either put the call in an `eval {...}` block or else call the subroutine in boolean context instead.

`Call to %s failed at %s. Attempted to use failure value at %s`
: This is the default exception that a `FAIL` throws when a failure value is captured in a scalar variable and later used in a non-boolean context. That means that the subroutine you called must have failed,

and you didn't check the return value for that failure, so when you tried to use that invalid value it killed your program. You should either put the original call in an `eval {...}` or else test the return value in a boolean context and avoid using it if it's false.

`Usage: FAIL_WITH $flag_opt, \%selector, @args`
The `FAIL_WITH` subroutine expects an optional flag, followed by a reference to a configuration hash, followed by a list or selector arguments. You gave it something else. See "Configurable Failure Contexts".

`Selector values must be sub refs`
You passed a configuration hash to `FAIL_WITH` that specified non− subroutines as possible `FAIL` handlers. Since non-subroutines can't possibly be handlers, maybe you forgot the `sub` keyword somewhere?

`Invalid option: %s = %s>`
The `FAIL_WITH` subroutine was passed a flag/selector pair, but the selector was not one of those allowed by the configuration hash.

`FAIL handler for package %s redefined`
A warning that the `FAIL` handler for a particular package was reconfigured more than once. Typically that's because the module was loaded in two places with difference configurations specified. You can't reasonably expect two different sets of behaviours from the one module within the one namespace.

CONFIGURATION AND ENVIRONMENT

Contextual::Return requires no configuration files or environment variables.

DEPENDENCIES

Requires version.pm and Want.pm.

INCOMPATIBILITIES

`LVALUE`, `RVALUE`, and `NVALUE` do not work correctly under the Perl debugger. This seems to be because the debugger injects code to capture the return values from subroutines, which interferes destructively with the optional final arguments that allow `LVALUE`, `RVALUE`, and `NVALUE` to cascade within a single return.

BUGS AND LIMITATIONS

No bugs have been reported.

AUTHOR

Damian Conway `<DCONWAY@cpan.org>`

LICENCE AND COPYRIGHT

DISCLAIMER OF WARRANTY

HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.