NAME

Want – A generalisation of "wantarray"

SYNOPSIS

```
use Want;
sub foo :lvalue {
    if    (want(qw'LVALUE ASSIGN')) {
      print "We have been assigned ", want('ASSIGN');
      lnoreturn;
    }
    elsif (want('LIST')) {
      rreturn (1, 2, 3);
    }
    elsif (want('BOOL')) {
      rreturn 0;
    }
    elsif (want(qw'SCALAR !REF')) {
      rreturn 23;
    }
    elsif (want('HASH')) {
      rreturn { foo => 17, bar => 23 };
    }
    return;  # You have to put this at the end to keep the compiler happy
}
```

DESCRIPTION

This module generalises the mechanism of the **wantarray** function, allowing a function to determine in some detail how its return value is going to be immediately used.

Top-level contexts:

The three kinds of top-level context are well known:

VOID

The return value is not being used in any way. It could be an entire statement like `foo();`, or the last component of a compound statement which is itself in void context, such as `$test || foo();`n. Be warned that the last statement of a subroutine will be in whatever context the subroutine was called in, because the result is implicitly returned.

SCALAR

The return value is being treated as a scalar value of some sort:

```
my $x = foo();
$y += foo();
print "123" x foo();
print scalar foo();
warn foo()->{23};
...etc...
```

LIST

The return value is treated as a list of values:

```
my @x = foo();
my ($x) = foo();
() = foo();           # even though the results are discarded
print foo();
bar(foo());           # unless the bar subroutine has a prototype
print @hash{foo()};   # (hash slice)
...etc...
```

Lvalue subroutines:

The introduction of **lvalue subroutines** in Perl 5.6 has created a new type of contextual information, which is independent of those listed above. When an lvalue subroutine is called, it can either be called in the ordinary way (so that its result is treated as an ordinary value, an **rvalue**); or else it can be called so that its result is considered updatable, an **lvalue**.

These rather arcane terms (lvalue and rvalue) are easier to remember if you know why they are so called. If you consider a simple assignment statement `left = right`, then the **l**eft-hand side is an **l**value and the **r**ight-hand side is an **r**value.

So (for lvalue subroutines only) there are two new types of context:

RVALUE

The caller is definitely not trying to assign to the result:

```
foo();
my $x = foo();
...etc...
```

If the sub is declared without the `:lvalue` attribute, then it will *always* be in RVALUE context.

If you need to return values from an lvalue subroutine in RVALUE context, you should use the `rreturn` function rather than an ordinary `return`. Otherwise you'll probably get a compile-time error in perl 5.6.1 and later.

LVALUE

Either the caller is directly assigning to the result of the sub call:

```
foo() = $x;
foo() = (1, 1, 2, 3, 5, 8);
```

or the caller is making a reference to the result, which might be assigned to later:

```
my $ref = \(foo());   # Could now have: $$ref = 99;

# Note that this example imposes LIST context on the sub call.
# So we're taking a reference to the first element to be
# returned _in list context_.
# If we want to call the function in scalar context, we can
# do it like this:
my $ref = \(scalar foo());
```

or else the result of the function call is being used as part of the argument list for *another* function call:

```
bar(foo());   # Will *always* call foo in lvalue context,
              # (provided that foo is an C<:lvalue> sub)
              # regardless of what bar actually does.
```

The reason for this last case is that bar might be a sub which modifies its arguments. They're rare in contemporary Perl code, but perfectly possible:

```
sub bar {
  $_[0] = 23;
}
```

(This is really a throwback to Perl 4, which didn't support explicit references.)

Assignment context:

The commonest use of lvalue subroutines is with the assignment statement:

```
size() = 12;
(list()) = (1..10);
```

A useful motto to remember when thinking about assignment statements is *context comes from the left*.

Consider code like this:

```
my ($x, $y, $z);
sub list () :lvalue { ($x, $y, $z) }
list = (1, 2, 3);
print "\$x = $x; \$y = $y; \$z = $z\n";
```

This prints `$x = ; $y = ; $z = 3`, which may not be what you were expecting. The reason is that the assignment is in scalar context, so the comma operator is in scalar context too, and discards all values but the last. You can fix it by writing `(list) = (1,2,3);` instead.

If your lvalue subroutine is used on the left of an assignment statement, it's in **ASSIGN** context. If ASSIGN is the only argument to `want()`, then it returns a reference to an array of the value(s) of the right-hand side.

In this case, you should return with the `lnoreturn` function, rather than an ordinary `return`.

This makes it very easy to write lvalue subroutines which do clever things:

```
use Want;
use strict;
sub backstr :lvalue {
  if (want(qw'LVALUE ASSIGN')) {
    my ($a) = want('ASSIGN');
    $_[0] = reverse $a;
    lnoreturn;
  }
  elsif (want('RVALUE')) {
    rreturn scalar reverse $_[0];
  }
  else {
    carp("Not in ASSIGN context");
  }
  return
}

print "foo -> ", backstr("foo"), "\n";        # foo -> oof
backstr(my $robin) = "nibor";
print "\$robin is now $robin\n";               # $robin is now robin
```

Notice that you need to put a (meaningless) return statement at the end of the function, otherwise you will get the error *Can't modify non-lvalue subroutine call in lvalue subroutine return*.

The only way to write that `backstr` function without using Want is to return a tied variable which is tied to a custom class.

Reference context:

Sometimes in scalar context the caller is expecting a reference of some sort to be returned:

```
print foo()->();     # CODE reference expected
print foo()->{bar};  # HASH reference expected
print foo()->[23];   # ARRAY reference expected
print ${foo()};      # SCALAR reference expected
print foo()->bar();  # OBJECT reference expected

my $format = *{foo()}{FORMAT} # GLOB reference expected
```

You can check this using conditionals like `if (want('CODE'))`. There is also a function `wantref()` which returns one of the strings "CODE", "HASH", "ARRAY", "GLOB", "SCALAR" or "OBJECT"; or the empty string if a reference is not expected.

Because `want('SCALAR')` is already used to select ordinary scalar context, you have to use

`want('REFSCALAR')` to find out if a SCALAR reference is expected. Or you could use `want('REF') eq 'SCALAR'` of course.

Be warned that `want('ARRAY')` is a **very** different thing from `wantarray()`.

Item count

Sometimes in list context the caller is expecting a particular number of items to be returned:

```
my ($x, $y) = foo();    # foo is expected to return two items
```

If you pass a number to the `want` function, then it will return true or false according to whether at least that many items are wanted. So if we are in the definition of a sub which is being called as above, then:

```
want(1) returns true
want(2) returns true
want(3) returns false
```

Sometimes there is no limit to the number of items that might be used:

```
my @x = foo();
do_something_with( foo() );
```

In this case, `want(2)`, `want(100)`, `want(1E9)` and so on will all return true; and so will `want('Infinity')`.

The `howmany` function can be used to find out how many items are wanted. If the context is scalar, then `want(1)` returns true and `howmany()` returns 1. If you want to check whether your result is being assigned to a singleton list, you can say `if (want('LIST', 1)) { ... }`.

Boolean context

Sometimes the caller is only interested in the truth or falsity of a function's return value:

```
if (everything_is_okay()) {
    # Carry on
}

print (foo() ? "ok\n" : "not ok\n");
```

In the following example, all subroutine calls are in BOOL context:

```
my $x = ( (foo() && !bar()) xor (baz() || quux()) );
```

Boolean context, like the reference contexts above, is considered to be a subcontext of SCALAR.

FUNCTIONS

want(SPECIFIERS)

This is the primary interface to this module, and should suffice for most purposes. You pass it a list of context specifiers, and the return value is true whenever all of the specifiers hold.

```
want('LVALUE', 'SCALAR');   # Are we in scalar lvalue context?
want('RVALUE', 3);          # Are at least three rvalues wanted?
want('ARRAY');      # Is the return value used as an array ref?
```

You can also prefix a specifier with an exclamation mark to indicate that you **don't** want it to be true

```
want(2, '!3');              # Caller wants exactly two items.
want(qw'REF !CODE !GLOB');  # Expecting a reference that
                            #   isn't a CODE or GLOB ref.
want(100, '!Infinity');     # Expecting at least 100 items,
                            #   but there is a limit.
```

If the *REF* keyword is the only parameter passed, then the type of reference will be returned. This is just a synonym for the `wantref` function: it's included because you might find it useful if you don't want to pollute your namespace by importing several functions, and to conform to Damian Conway's suggestion in RFC 21.

Finally, the keyword *COUNT* can be used, provided that it's the only keyword you pass. Mixing COUNT with other keywords is an error. This is a synonym for the `howmany` function.

A full list of the permitted keyword is in the **ARGUMENTS** section below.

rreturn
: Use this function instead of `return` from inside an lvalue subroutine when you know that you're in RVALUE context. If you try to use a normal `return`, you'll get a compile-time error in Perl 5.6.1 and above unless you return an lvalue. (Note: this is no longer true in Perl 5.16, where an ordinary return will once again work.)

lnoreturn
: Use this function instead of `return` from inside an lvalue subroutine when you're in ASSIGN context and you've used `want('ASSIGN')` to carry out the appropriate action.

  If you use `rreturn` or `lnoreturn`, then you have to put a bare `return;` at the very end of your lvalue subroutine, in order to stop the Perl compiler from complaining. Think of it as akin to the `1;` that you have to put at the end of a module. (Note: this is no longer true in Perl 5.16.)

howmany()
: Returns the *expectation count*, i.e. the number of items expected. If the expectation count is undefined, that indicates that an unlimited number of items might be used (e.g. the return value is being assigned to an array). In void context the expectation count is zero, and in scalar context it is one.

  The same as `want('COUNT')`.

wantref()
: Returns the type of reference which the caller is expecting, or the empty string if the caller isn't expecting a reference immediately.

  The same as `want('REF')`.

EXAMPLES

```
use Carp 'croak';
use Want 'howmany';
sub numbers {
    my $count = howmany();
    croak("Can't make an infinite list") if !defined($count);
    return (1..$count);
}
my ($one, $two, $three) = numbers();


use Want 'want';
sub pi () {
    if    (want('ARRAY')) {
        return [3, 1, 4, 1, 5, 9];
    }
    elsif (want('LIST')) {
        return (3, 1, 4, 1, 5, 9);
    }
    else {
        return 3;
    }
}
print pi->[2];      # prints 4
print ((pi)[3]);    # prints 1
```

ARGUMENTS

The permitted arguments to the `want` function are listed below. The list is structured so that sub-contexts appear below the context that they are part of.

- VOID
- SCALAR
  - REF
    - REFSCALAR
    - CODE
    - HASH
    - ARRAY
    - GLOB
    - OBJECT
  - BOOL
- LIST
  - COUNT
  - <number>
  - Infinity
- LVALUE
  - ASSIGN
- RVALUE

EXPORT

The `want` and `rreturn` functions are exported by default. The `wantref` and/or `howmany` functions can also be imported:

```
use Want qw'want howmany';
```

If you don't import these functions, you must qualify their names as (e.g.) `Want::wantref`.

INTERFACE

This module is still under development, and the public interface may change in future versions. The `want` function can now be regarded as stable.

I'd be interested to know how you're using this module.

SUBTLETIES

There are two different levels of **BOOL** context. *Pure* boolean context occurs in conditional expressions, and the operands of the `xor` and `!`/`not` operators. Pure boolean context also propagates down through the `&&` and `||` operators.

However, consider an expression like `my $x = foo() && "yes"`. The subroutine is called in *pseudo*−boolean context − its return value isn't **entirely** ignored, because the undefined value, the empty string and the integer 0 are all false.

At the moment `want('BOOL')` is true in either pure or pseudo boolean context. Let me know if this is a problem.

BUGS

```
* Doesn't work from inside a tie-handler.
```

AUTHOR

Robin Houston, <robin@cpan.org>

Thanks to Damian Conway for encouragement and good suggestions, and Father Chrysostomos for a patch.

SEE ALSO

- ‘‘wantarray’’ in perlfunc
- Perl6 RFC 21, by Damian Conway. http://dev.perl.org/rfc/21.html

COPYRIGHT