NAME

B::CC - Perl compiler's optimized C translation backend

SYNOPSIS

```
perl -MO=CC[,OPTIONS] foo.pl
```

DESCRIPTION

This compiler backend takes Perl source and generates C source code corresponding to the flow of your program. In other words, this backend is somewhat a "real" compiler in the sense that many people think about compilers. Note however that, currently, it is a very poor compiler in that although it generates (mostly, or at least sometimes) correct code, it performs relatively few optimisations. This will change as the compiler develops. The result is that running an executable compiled with this backend may start up more quickly than running the original Perl program (a feature shared by the **C** compiler backend--see *B::C*) and may also execute slightly faster. This is by no means a good optimising compiler--yet.

OPTIONS

If there are any non-option arguments, they are taken to be names of objects to be saved (probably doesn't work properly yet). Without extra arguments, it saves the main program.

-ofilename

Output to filename instead of STDOUT

-v

Verbose compilation (currently gives a few compilation statistics).

--

Force end of options

-uPackname

Force apparently unused subs from package Packname to be compiled. This allows programs to use eval "foo()" even when sub foo is never seen to be used at compile time. The down side is that any subs which really are never used also have code generated. This option is necessary, for example, if you have a signal handler foo which you initialise with `$SIG{BAR} = "foo"`. A better fix, though, is just to change it to `$SIG{BAR} = \&foo`. You can have multiple **-u** options. The compiler tries to figure out which packages may possibly have subs in which need compiling but the current version doesn't do it very well. In particular, it is confused by nested packages (i.e. of the form `A::B`) where package `A` does not contain any subs.

-mModulename

Instead of generating source for a runnable executable, generate source for an XSUB module. The boot_Modulename function (which DynaLoader can look for) does the appropriate initialisation and runs the main part of the Perl source that is being compiled.

-D

Debug options (concatenated or separate flags like `perl -D`).

-Dr

Writes debugging output to STDERR just as it's about to write to the program's runtime (otherwise writes debugging info as comments in its C output).

-DO

Outputs each OP as it's compiled

-Ds

Outputs the contents of the shadow stack at each OP

-Dp

Outputs the contents of the shadow pad of lexicals as it's loaded for each sub or the main program.

-Dq

Outputs the name of each fake PP function in the queue as it's about to process it.

-Dl

Output the filename and line number of each original line of Perl code as it's processed (`pp_nextstate`).

-Dt

Outputs timing information of compilation stages.

-f

Force optimisations on or off one at a time.

-ffreetmps-each-bblock

Delays FREETMPS from the end of each statement to the end of the each basic block.

-ffreetmps-each-loop

Delays FREETMPS from the end of each statement to the end of the group of basic blocks forming a loop. At most one of the freetmps-each-* options can be used.

-fomit-taint

Omits generating code for handling perl's tainting mechanism.

-On

Optimisation level (n = 0, 1, 2, ...). **-O** means **-O1**. Currently, **-O1** sets **-ffreetmps-each-bblock** and **-O2** sets **-ffreetmps-each-loop**.

EXAMPLES

```
perl -MO=CC,-O2,-ofoo.c foo.pl
perl cc_harness -o foo foo.c
```

Note that `cc_harness` lives in the `B` subdirectory of your perl library directory. The utility called `perlcc` may also be used to help make use of this compiler.

```
perl -MO=CC,-mFoo,-oFoo.c Foo.pm
perl cc_harness -shared -c -o Foo.so Foo.c
```

BUGS

Plenty. Current status: experimental.

DIFFERENCES

These aren't really bugs but they are constructs which are heavily tied to perl's compile-and-go implementation and with which this compiler backend cannot cope.

Loops

Standard perl calculates the target of "next", "last", and "redo" at run-time. The compiler calculates the targets at compile-time. For example, the program

```
sub skip_on_odd { next NUMBER if $_[0] % 2 }
NUMBER: for ($i = 0; $i < 5; $i++) {
    skip_on_odd($i);
    print $i;
}
```

produces the output

```
    024
```

with standard perl but gives a compile-time error with the compiler.

Context of ".."

The context (scalar or array) of the ".." operator determines whether it behaves as a range or a flip/flop. Standard perl delays until runtime the decision of which context it is in but the compiler needs to know the context at compile-time. For example,

```
    @a = (4,6,1,0,0,1);
    sub range { (shift @a)..(shift @a) }
    print range();
    while (@a) { print scalar(range()) }
```

generates the output

```
    456123E0
```

with standard Perl but gives a compile-time error with compiled Perl.

Arithmetic

Compiled Perl programs use native C arithmetic much more frequently than standard perl. Operations on large numbers or on boundary cases may produce different behaviour.

Deprecated features

Features of standard perl such as `$[` which have been deprecated in standard perl since Perl5 was released have not been implemented in the compiler.

AUTHOR

Malcolm Beattie, `mbeattie@sable.ox.ac.uk`