# Ace: a syntax-driven C preprocessor

*James Gosling*

*July, 1989*

## Abstract

*This document presents the **ace** preprocessor for C programs. Unlike **cpp**, which operates on characters, **ace** operates on syntax trees. The user specifies syntax trees which are used as templates against which program fragments are matched. Positive matches cause trees to be rewritten. **Ace** can be used as a special-purpose optimizer that can be controlled by the programmer.*

## 1. Introduction

Ace is a preprocessor for C programs, a sort of “macro processor” in the spirit of cpp. Unlike cpp, ace “macros” do not operate on strings of characters, they operate on syntax trees. Instead of macros, ace has *rules*. A rule consists of a pattern in the form of a syntax tree, and a replacement, also a syntax tree. These rules cause instances of the pattern to be replaced in the program tree. Ace reads in a C source program, constructs its syntax tree, performs any replacements, and writes the tree out as a C program.

The design of ace was motivated by a desire to perform transformations on algorithms to improve their performance, without impacting their readability and maintainability. As an example of the kind of thing it can do, consider this code fragment:

```
for (i = 0; i<10; i++)
      if(da > 0) A[i] ++;
      else A[i] --;
```

The test in the inner loop, `da>0`, is loop-invariant: it doesn’t change from one trip through the loop to another. This loop can be rewritten as:

```
if (da > 0)
      for (i = 0; i<10; i++) A[i]++;
else
      for (i = 0; i<10; i++) A[i]--;
```

Eliminating this is a form of code motion that no compilers use since it leads to an exponential growth in code size, but in some cases it is justified.The exponential code growth comes from the fact that much of the body of the loop is replicated. Each one of these invariant tests that is removed from a loop body doubles the size of the code. But in some circumstances, like the inner loop of a vector drawing routine, this cost in code expansion is gladly paid.

Often the way that people deal with optimizations like this is that they expand the code by hand with a text editor. But once this is done, the original code is destroyed and the relationships between the parts is obscured. If you wanted to change, for example, the upper bound on the loop from 10 to 11, you would now have to change it in two places rather than one. When the loop body becomes large, and the number of such special cases becomes large, doing this transformation by hand becomes a major undertaking. Using ace, the second piece of code can be generated from the first by prefixing it by one line, like so:

```
$pullout(da > 0)
      for (i = 0; i<10; i++)
            if(da > 0) A[i] ++;
            else A[i] --;
```

## 2. Rules

*Ace* understands the syntax of C with a few additions. To avoid name clashes, *ace* considers $ to be a legal character in identifiers. By convention, names specific to *ace* start with $. The most important addition is the *$replace* statement. It looks like this:

```
$replace statement1 $with statement2
```

This defines a rule that causes all occurrences of *statement1* to be replaced by *statement2*. Statement1 is a *template*. Since expressions are syntactically statements in C, $replace can be used to define replacements for expressions as well as statements:

```
$replace sqrt(4); $with 2;
a = sqrt(4);
```

`sqrt(4)` in the second line will be replaced by `2`. *$replace* definitions are applied to the rest of the file. When multiple templates match a tree, the one from the earliest *$replace* statement applies.

The templates can contain unbound meta variables that match anything. These are the symbols $0, $1, $2 ... For example:

```
$replace !($0<$1); $with $0>=$1;
if (!(a<b+3)) ....
```

`!(a<b+3)` will be replaced by `a>=b+3`. Sometimes it's necessary to restrict the matches of these meta variables. One restriction is to trees that are side-effect free. Such matches are indicated with $f0, $f1, $f2... For example:

```
$replace $f0 = $0; $with $0;
a = a;
*p++ = *p++;
```

The first assignment statement, `a=a` would be replaced by `a`, since evaluating `a` has no side effects. The second assignment wouldn't be replaced since `p++` has a side effect. *Ace* is reasonably clever about statements and will eliminate those that have no side effects, so replacing `a=a` in a statement context with `a` causes the whole statement to be eliminated. But `b=2*(a=a)` would become `b=2*a`.

*$LET* is a special function that *ace* understands:

$\mathtt{\$LET(a_0,a_1,a_2,a_3,...,a_n)}$

This temporarily defines rules that replace $\mathtt{a_0}$ with $\mathtt{a_1}$, $\mathtt{a_2}$ with $\mathtt{a_3}$, ... in $\mathtt{a_n}$. For example `$LET(a, 1, a+b)` would expand to `1+b`.

As a useful piece of syntactic sugar, *ace* extends the C language with *prefix statements*. A prefix statement is just a statement that has been prefixed by something that looks like a procedure call. The statement becomes a last argument to the procedure call. This procedure call will normally be transformed into a statement by *ace* rules. These prefix statements are defined with the *$defprefix* procedure:

```
$defprefix($let, $LET);
```

This defines *$let* to be a prefix statement that is replaced by a call of the $LET function:

```
$let(a,1) {
      b = a+1;
      print(a);
}
```

becomes

```
{
      b = 2;
      print(1);
}
```

Unlike *cpp*, one doesn't have to insert parenthesis all over the place in ace rules:

```
$replace angle($0); $with $0->angle;
```

When *ace* rewrites `angle(*p)`, `$0` matches `*p`. When the syntax tree is finally printed, *ace* correctly inserts parenthesis based on operator priorities to yield `(*p)->angle`.

## 3. Time/Space tradeoffs

*Ace* has a facility that allows you to make time/space tradeoffs:

$tradeoff($code_1$,$code_2$)

picks either $\text{code}_1$ or $\text{code}_2$ depending on a time/space tradeoff. Presumably, $\text{code}_1$ and $\text{code}_2$ perform the same computation, only in different ways. *Ace* will estimate the time used by each code fragment and the space used. To aid in its computation of a time estimate, it needs to know the probability that branches will go one way or another and it needs to know the expected number of trips through a loop. $Replace is used to tell *ace* the probability that a boolean expression will be true:

```
$replace $P(e); $with c;
```

This says that the probability that *e* will be true is *c*. These probability specifications are used in **if** and **switch** statements to determine the probability of execution of each clause. If *ace* cannot determine the probability of some expression, it will assume that all clauses are equally likely.

The expected number of trips through a loop is specified by prefixing it with $trips:

```
$trips(100)
      for(i=0; i<100; i++) { ... }
```

This tells *ace* that the for loop is expected to be executed about 100 times.

Based on this information, and two parameters, *ace* will pick one of the two code fragment parameters of $tradeoff to replace $tradeoff. The two parameters are *pthresh* and *mingain*. *Pthresh* is a probability threshold: If the probability of executing a particular $tradeoff exceeds *pthresh* then the time-efficient code fragment will be chosen, otherwise the space-efficient fragment will be chosen. *mingain* specifies a minimum percentage time gain. If the code fragment chosen by *pthresh* doesn't gain at least *mingain* percent in time, the space-efficient code fragment will be chosen.

## 4. Rule Application Order

Once the source has been parsed, *Ace* applies the rules that have been defined. They are applied by traversing the parse tree from the root. It attempts to apply rules to a node *both* before *and* after the rules have been applied to its subnodes. Rules are applied before so that rules which change the ruleset (e.g. those that use let) behave properly. They're also applied after in case the transformed subnodes expose new opportunities for rule application.

This can cause some subtle interactions. Consider the following:

```
$replace log2(2); $with 1;
$replace constant($c0); $with 1;
$replace constant($0); $with 0;
constant(1)
constant(a)
constant(log2(2))
```

The intent of the constant rule is that it should evaluate to true if it's argument is a constant, and false otherwise. Because of the ordering of the definitions, this should be so: If the argument is a constant, the second rule will be applied, yielding true. If it isn't, the third will be applied, yielding false. Constant(1) evaluates to true, and constant(a) evaluates to false. But constant(log2(2)) evaluates to false because when the rules are applied before reducing the argument to constant, the third rule is used since log2(2) isn't a constant. There is a way around this:

```
$replaceafter constant($0); with 0;
```

If a rule is defined with $replaceafter rather than with $replace, it will only be applied to a node after its arguments have been reduced.

## 5. Building on ace

Using ace, we can define DeMorgan's law:

```
      $replace ! ($0 && $1); $with ! $0 || !$1;
      $replace ! ($0 || $1); $with ! $0 && !$1;
```

Then there are a number of rules that are used in conjunction with these:

```
      $replace ! ($0 == $1);        $with $0 != $1;
      $replace ! ($0 != $1);        $with $0 == $1;
      $replace ! ($0 >= $1);        $with $0 < $1;
      $replace ! ($0 <= $1);        $with $0 > $1;
      $replace ! ($0 > $1);         $with $0 <= $1;
      $replace ! ($0 < $1);         $with $0 >= $1;
      $replace ! !$0;               $with $0;
```

Now we can define a more subtle rule:

```
      $replace $assume($0, $1);
      $with $let($0, 1, !$0, 0, $1);
```

This rule causes code fragment $1 to be compiled, assuming that $0 is true, and that !$0 is false:

```
      $defprefix($ASSUME, $assume);
      $ASSUME(a<0) {
            if (a>=0) print("true");
            else print ("false");
      }
```

This is transformed into just `print("false")`. $Assume will replace `a<0` with 1, and `!(a<0)` with 0. Other rules ensure that `!(a<0)` is replaced by `a>=0`, which is replaced by `0`. The *if* now has a constant to test, so the true clause is eliminated. There are many special cases of the assume rule that can be defined. Because of the ordering rule of template matching, they have to precede the general rule:

```
      $replace $assume($0 == $1, $2);
      $with $let($0, $1, $2);
      $replace $assume($0 < $1, $2);
      $with $let($0<$1, 1,
            $0>=$1, 0,
            $0<=$1, 1,
            $0==$1, 0,
            $0 != $1, 1, $2);
      $replace $assume($0 > $1, $2);
      $with $let($0>$1, 1,
            $0<=$1, 0,
            $0>=$1, 1,
            $0==$1, 0,
            $0 != $1, 1, $2);
      $replace $assume($0 && $1, $2);
      $with $assume($0, $assume($1, $2));
```

Using these, we can now define the *$pullout* prefix that was used at the beginning of this description:

```
      $defprefix($pullout, $pulloute);
      $defprefix($LET, $let);
      $replace $pulloute($0, $2);
      $with       if($0)
                        $assume($0, $2);
                  else
                        $assume(!$0, $2);
```

In other words, to pull a test out of a code fragment, perform the test, and when it's true execute the code assuming that the test is true, and when it's false, execute the code assuming that it's false. Repeating the example from the beginning of this paper:

```
      $pullout(da > 0)
            for (i = 0; i<10; i++)
                  if(da > 0) A[i] ++;
                  else A[i] --;
```

This gets expanded to:

```
if (da>0)
      $ASSUME(da > 0)
            for (i = 0; i<10; i++)
                  if(da > 0) A[i] ++;
                  else A[i] --;
else  $ASSUME(!(da > 0))
            for (i = 0; i<10; i++)
                  if(da > 0) A[i] ++;
                  else A[i] --;
```

The ASSUME clauses cause this to become:

```
if (da>0)
            for (i = 0; i<10; i++)
                  if(1) A[i] ++;
                  else A[i] --;
else
            for (i = 0; i<10; i++)
                  if(0) A[i] ++;
                  else A[i] --;
```

And constant collapsing eliminates the inner **if**s, yielding:

```
if (da>0)
            for (i = 0; i<10; i++)
                  A[i] ++;
else
            for (i = 0; i<10; i++)
                  A[i] --;
```

Tradeoff() can be used to make $pullout() much more powerful:

```
$replace $pulloute($0, $2);
$with       $tradeoff($0      ? $assume($0, $2)
                              : $assume(!$0, $2),
                        $2);
```

This pulls $0 out of $2 only if there is a useful performance gain. **Note**: *ace* treats ? and **if** identically.

## 6. More mundane uses

Ace can be used much like cpp to define procedures that are expanded inline, with the added attraction that it’s easy to define special cases for parameters that are known at compile time:

```
/* bool(a,b,op) executes a boolean operation specified by op */
$replace bool($0,$1,0); $with $0|$1;
$replace bool($0,$1,1); $with $0&$1;
```

It can, of course, match parameters other than constants:

```
$replace Get_Context($0, CTX_CLIP); $with $0->CTX_CLIP;
```

It can provide default parameters to procedure calls:

```
$replace atan2($0); $with atan2($0, 1);
```

It can be used to define iterators for special data types:

```
/* shape iterator */
$defprefix($scanshape, $scanshapee);
$replace $scanshapee($0, $1);/* (shape, code) */
$with {
      register ENTRY *sptr = Get_Shape($0, SHAPE_DATA);
      short       x0,
                  y0,
                  x1,
                  y1;
      while (*sptr != Y_EOL) {
            y0 = *sptr++;
            y1 = *sptr++;
            while (*sptr != X_EOL) {
```

```
                        x0 = *sptr++;
                        x1 = *sptr++;
                        $1;
                }
                sptr++;
        }
}
```

This example uses the special prefix syntax so that it can be invoked this way:

```
        $scanshape(thisshape) {
                printf("%d, %d, %d, %d\n", x0, y0, x1, y1);
                FillRectangle(x0, y0, x1, y1);
        }
```

## 7. Acknowledgements

A special thanks to Patrick Naughton for being a guinea pig user. And to the whole “Shapes” team that put up with mysterious things breaking.

## Appendix 1. Invoking *ace*

```
        ace [-time] [-space] [-lnc] [-nln] [-pthresh n]
                [-mingain m] [-Qpath path] [-o ofile] ifile
```

| | |
|---|---|
| **-pthresh** *n* | Sets the *pthresh* parameter to n. See the section on time/space tradeoffs |
| **-time** | Optimize for time. It’s the same as **-pthresh 0**. |
| **-space** | Optimize for space. It’s the same as **-pthresh 1**. |
| **-lnc** | Line numbers as comments: each line generated by *ace* will be prefixed with a comment that tells what line of the input file it came from. This is a good switch to use for debugging since dbx will step through the expanded output, but you’ll be able to find the code in the original source. |
| **-nln** | No line numbers. This should be used if you want to read the code generated by *ace*. It removes the clutter left by line numbers. |
| **-mingain** *m* | Sets the *mingain* parameter to *m*. See the section on time/space tradeoffs. |
| **-Qpath** *path* | Causes *ace* to look in *path* for *cpp*. Normally it just looks in /lib and /usr/lib. |
| **-o** *ofile* | Sends the generated output to *ofile*. The default is standard out. *Ofile* will be unlinked before it is created, and it will be created with mode 444, to prevent accidental editing. |
| **-ifc** | Includes comments after each **if** and **else** that indicate what’s true and what’s false, according to containing if statements. |

Ace pipes its input through *cpp* and takes all parameters that *cpp* would accept and passes them on to it.

## Appendix 2. A Large Example

As an example of how *ace* can be used in a real-world example, here is a routine for drawing vectors using Bresenham’s algorithm. It is almost exactly the same as the vector routine that appears in the Shapes library, except that the code to support clipping has been eliminated. But is does handle several framebuffer depths (1, 8 and 32 bits per pixel), plane masks, and all 16 rasterop codes. Normally the inner loop is written out many times for the various special cases. Here, it is written once, and *ace* is used to generate all of the special cases:

```
sh_fb_VecPt(ras, X1, Y1, X2, Y2)
    RASTER      ras;
{
    register short count;
    register int err;
    register int erra;
    register int errb;
    int         plane_enable = FB_plane_enable,
                dx, dy, left, lineiny;
    if (Y1 > Y2) {
         swap_coord(X1, X2, left);
         swap_coord(Y1, Y2, left);
    }
    dy = Y2 - Y1;
    dx = X2 - X1;
    if (left = (dx < 0))
         dx = -dx;
    if (lineiny = (dy > dx)) {
         count = dy + 1; erra = dx << 1; errb = dy << 1;
         err = left ? 1 - dy : -dy;
    } else {
         count = dx + 1; erra = dy << 1; errb = dx << 1;
         err = -dx;
    }
    $switchout(DEPTH, ras->RAS_DEPTH, SH_SUPPORTED_DEPTHS) {
         register int bpsl = ras->RAS_LINEBYTES;
         PIXCOLOR(color, FB_col, DEPTH);
         PIXROP(ropcode, (int) FB_rop, color, DEPTH);
         PIXPTR(pix, DEPTH);
         PIXMASK(mask, DEPTH);
         initpixelpointer_no(ras, pix, mask, X1, Y1, DEPTH);
         --count;          /* $repeat(count) generates count+1 loops */
         $fastrops(ropcode, DEPTH)
             $alwayspulloutiff((FB_disp & FB_DRAW_PLANES) == 0,
                                 plane_enable == ~0)
             $alwayspullout(erra != 0)
             $alwayspullout(lineiny == 0)
             $alwayspullout(left == 0)
             $repeat(count) {
             writepixel(pix, mask, DEPTH,
                         ropcode, color, plane_enable);
             if (lineiny == 0)
                  if (left == 0)
                      RIGHTSTEP(pix, mask, DEPTH);
                  else
                      LEFTSTEP(pix, mask, DEPTH);
             else
                  DOWNSTEP_STRIDE(bpsl, pix, DEPTH);
             if (erra != 0)
                  if ((err += erra) >= 0) {
                      err -= errb;
                      if (lineiny != 0)
                           if (left == 0)
                               RIGHTSTEP(pix, mask, DEPTH);
                           else
                               LEFTSTEP(pix, mask, DEPTH);
                      else
                           DOWNSTEP_STRIDE(bpsl, pix, DEPTH);
                  }
         }
    }
}
```

Running this through *ace* yields a 20 page source file that contains expanded code for all the special cases. As you can see, the inner loops are all very tight. The following listing has been substantially abbreviated. The expansion of the `$repeat` macro is especially interesting: it is machine dependent. In this case it expands to `do { ... } while (--count != -1)`, which is compiled on a 68020 into a dbra instruction.

```
int sh_fb_Vect(ras, X1, Y1, X2, Y2)
  RASTER ras; {
  register short count;
  register int err;
  register int erra;
  register int errb;
  int plane_enable = sh_fb_attrs.plane_enable, dx, dy, left, lineiny;
  if (Y1 > Y2) {
      left = X1;
      X1 = X2;
      X2 = left;
      left = Y1;
      Y1 = Y2;
      Y2 = left; }
  dy = Y2 - Y1;
  dx = X2 - X1;
  if (left = dx < 0)
    dx =  - dx;
  if (lineiny = dy > dx) {
      count = dy + 1;
      erra = dx << 1;
      errb = dy << 1;
      err = left ? 1 - dy :  - dy; }
  else {
      count = dx + 1;
      erra = dy << 1;
      errb = dx << 1;
      err =  - dx; }
  switch (ras->RAS_DEPTH) {
  case 1: {
      register int bpsl = ras->RAS_LINEBYTES;
      int color = sh_fb_attrs.col;
      int ropcode = color ? mono_remap1[(int) sh_fb_attrs.rop] :
                        mono_remap0[(int) sh_fb_attrs.rop];
      register unsigned short *pix;
      register unsigned short mask;
      pix = (unsigned short * ) (ras->RAS_DATA + (short) ras->RAS_LINEBYTES*
        (short) Y1 + (X1 >> 3 & -2));
      mask = 32768 >> (X1 & 15);
      --count;
      switch (ropcode) {
      case 14:
        if (erra != 0) {
          if (lineiny == 0) {
            if (left == 0)
```

*Monochrome (1 bit deep), going right, x is the major axis, the line is neither horizontal nor vertical, and the ropcode is SRC.*

```
              do {
                  *pix |= mask;
                  if ((mask >>= 1) == 0) {
                      mask = 32768;
                      pix++; }
                  if ((err += erra) >= 0) {
                      err -= errb;
                      pix = (unsigned short * ) ((int) pix + bpsl); } }
              while (--count != -1);
            else
```

*Monochrome (1 bit deep), going left, x is the major axis, the line is neither horizontal nor vertical, and the ropcode is SRC.*

```
              do {
                  *pix |= mask;
                  if ((mask = (unsigned short) (mask << 1)) == 0) {
                      mask = 1;
                      pix--; }
                  if ((err += erra) >= 0) {
                      err -= errb;
                      pix = (unsigned short * ) ((int) pix + bpsl); } }
              while (--count != -1);
```

```
          else
```

*Monochrome (1 bit deep), going right, y is the major axis, the line is neither horizontal nor vertical, and the ropcode is SRC.*

```
            if (left == 0)
              do {
                  *pix |= mask;
                  pix = (unsigned short * ) ((int) pix + bpsl);
                  if ((err += erra) >= 0) {
                      err -= errb;
                      if ((mask >>= 1) == 0) {
                          mask = 32768;
                          pix++; } } }
              while (--count != -1);
            else
              do {
                  *pix |= mask;
                  pix = (unsigned short * ) ((int) pix + bpsl);
                  if ((err += erra) >= 0) {
                      err -= errb;
                      if ((mask = (unsigned short) (mask << 1)) == 0) {
                          mask = 1;
                          pix--; } } }
              while (--count != -1);
        else
          if (lineiny == 0) {
            if (left == 0)
              do {
                  *pix |= mask;
                  if ((mask >>= 1) == 0) {
                      mask = 32768;
                      pix++; } }
              while (--count != -1);
            else
              do {
                  *pix |= mask;
                  if ((mask = (unsigned short) (mask << 1)) == 0) {
                      mask = 1;
                      pix--; } }
              while (--count != -1);
          else
            do {
                *pix |= mask;
                pix = (unsigned short * ) ((int) pix + bpsl); }
            while (--count != -1);
        break;
.
.
.
  case 8: {
      register int bpsl = ras->RAS_LINEBYTES;
      register int color = sh_fb_attrs.col;
      int ropcode = (int) sh_fb_attrs.rop;
      register unsigned char *pix;
      pix = ras->RAS_DATA + (short) ras->RAS_LINEBYTES*(short) Y1 + X1;
      --count;
      switch (ropcode) {
      case 12:
        if ((sh_fb_attrs.disp & 1) == 0) {
          if (erra != 0) {
            if (lineiny == 0) {
              if (left == 0)
```

*8 bit deep pixels, going right, x is the major axis, the line is neither horizontal nor vertical, all planes enabled, and the ropcode is SRC.*

```
                do {
                    *pix = color;
                    ++pix;
                    if ((err += erra) >= 0) {
                        err -= errb;
                        pix += bpsl; } }
                while (--count != -1);
              else
```

*8 bit deep pixels, going left, x is the major axis, the line is neither horizontal nor vertical, all planes enabled, and the ropcode is SRC.*

```
                do {
                    *pix = color;
                    --pix;
                    if ((err += erra) >= 0) {
                        err -= errb;
                        pix += bpsl; } }
                while (--count != -1);
.
.
.           else
             if (lineiny == 0) {
               if (left == 0)
```

*8 bit deep pixels, going right, x is the major axis, the line is horizontal, not all planes are enabled, and the ropcode is SRC.*

```
                do {
                    *pix = color & plane_enable | *pix & ~plane_enable;
                    ++pix; }
                while (--count != -1);
              else
                do {
                    *pix = color & plane_enable | *pix & ~plane_enable;
                    --pix; }
                while (--count != -1);
            else
              do {
                  *pix = color & plane_enable | *pix & ~plane_enable;
                  pix += bpsl; }
              while (--count != -1);
        break;
.
.
.
  case 32: {
      register int bpsl = ras->RAS_LINEBYTES;
      register int color = sh_fb_attrs.col;
      int ropcode = (int) sh_fb_attrs.rop;
      register unsigned char *pix;
      pix = ras->RAS_DATA + (short) ras->RAS_LINEBYTES*(short) Y1 + X1*
        4;
      --count;
      switch (ropcode) {
      case 12:
        if ((sh_fb_attrs.disp & 1) == 0) {
          if (erra != 0) {
            if (lineiny == 0) {
              if (left == 0)
```

*32 bit deep pixels, going right, x is the major axis, the line is neither horizontal nor vertical, all planes enabled, and the ropcode is SRC.*

```
                do {
                    *(int * ) pix = color;
                    pix += 4;
                    if ((err += erra) >= 0) {
                        err -= errb;
                        pix += bpsl; } }
                while (--count != -1);
.
.
.
        else
            if (lineiny == 0) {
              if (left == 0)
```

*32 bit deep pixels, going right, x is the major axis, the line is horizontal, all planes enabled, and the ropcode is SRC.*

```
                do {
                    *(int * ) pix = color;
                    pix += 4; }
                while (--count != -1);
              else
```

*32 bit deep pixels, going left, x is the major axis, the line is horizontal, all planes enabled, and the ropcode is SRC.*

```
                do {
                    *(int * ) pix = color;
                    pix -= 4; }
                while (--count != -1);
            else
```

*32 bit deep pixels, going neither left nor right, y is the major axis, the line is vertical, all planes enabled, and the ropcode is SRC.*

```
              do {
                  *(int * ) pix = color;
                  pix += bpsl; }
              while (--count != -1);
```