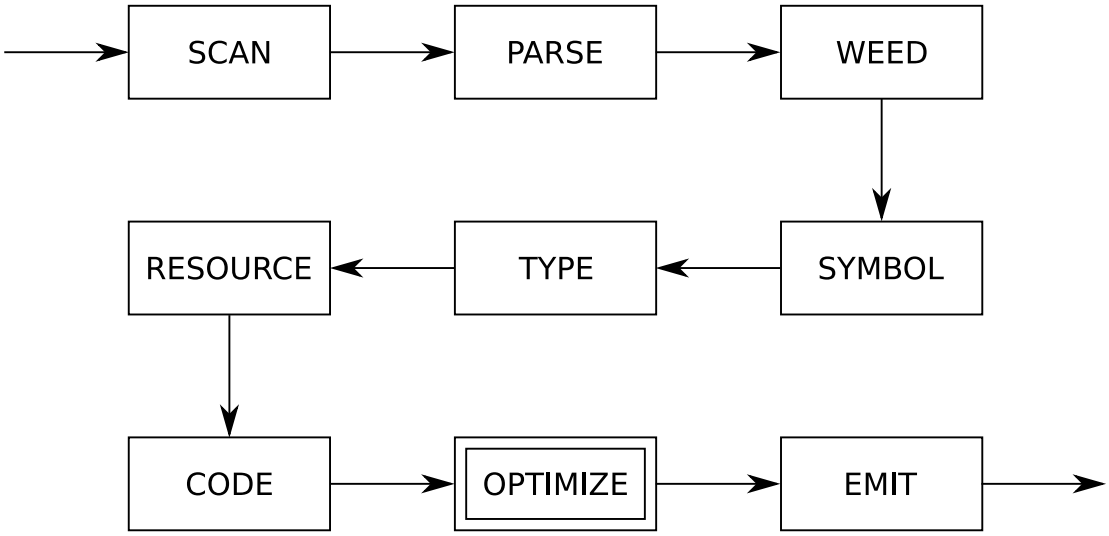
Optimization



The *optimizer* focuses on:

- reducing the execution time; or
- reducing the code size; or
- reducing the power consumption (new).

These goals often conflict, since a larger program may in fact be faster.

The best optimizations achieve both goals.

Optimizations for space:

- were historically very important, because memory was small and expensive;
- when memory became large and cheap, optimizing compilers traded space for speed; but
- then Internet bandwidth is small and expensive, so Java compilers optimize for space,
- today Internet bandwidth is larger and cheaper, so we optimize for speed again.

$\Rightarrow$ Optimizations driven by economy!

Optimizations for speed:

- were historically very important to gain acceptance for high-level languages;
- are still important, since the software always strains the limits of the hardware;
- are challenged by ever higher abstractions in programming languages; and
- must constantly adapt to changing microprocessor architectures.

Optimizations may take place:

- at the source code level;
- in an intermediate representation;
- at the binary machine code level; or
- at run-time (e.g. JIT compilers).

An aggressive optimization requires many small contributions from all levels.

Should you program in "Optimized C"?

If you want a fast C program, should you use `LOOP #1` or `LOOP #2`?

```
/* LOOP #1 */
for (i = 0; i < N; i++) {
    a[i] = a[i] * 2000;
    a[i] = a[i] / 10000;
}

/* LOOP #2 */
b = a;
for (i = 0; i < N; i++) {
    *b = *b * 2000;
    *b = *b / 10000;
    b++;
}
```

What would the expert programmer do?

If you said `LOOP #2` ... you were wrong!

`LOOP`	opt. level	SPARC	MIPS	Alpha
`#1` (array)	no opt	20.5	21.6	7.85
`#1` (array)	opt	8.8	12.3	3.26
`#1` (array)	super	7.9	11.2	2.96
`#2` (ptr)	no opt	19.5	17.6	7.55
`#2` (ptr)	opt	12.4	15.4	4.09
`#2` (ptr)	super	10.7	12.9	3.94

- Pointers confuse most C compilers; don't use pointers instead of array references.
- Compilers do a good job of register allocation; don't try to allocate registers in your C program.
- In general, write clear C code; it is easier for both the programmer and the compiler to understand.

Optimization in JOOS:

```
c = a*b+c;
if (c<a) a=a+b*113;
while (b>0) {
  a=a*c;
  b=b-1;
}
```

```
iload_1
iload_2
imul
iload_3
iadd
dup
istore_3
pop
iload_3
iload_1
if_icmplt true_1
iconst_0
goto stop_2
true_1:
iconst_1
stop_2:
ifeq stop_0
iload_1
iload_2
ldc 113
imul
iadd
dup
istore_1
pop
stop_0:
start_3:
iload_2
iconst_0
if_icmpgt true_5
iconst_0
goto stop_6
true_5:
iconst_1
stop_6:
ifeq stop_4
iload_1
iload_3
imul
dup
istore_1
pop
iload_2
iconst_1
isub
dup
istore_2
pop
goto start_3
stop_4:
```

```
iload_1
iload_2
imul
iload_3
iadd
istore_3
iload_3
iload_1
if_icmpge stop_0
iload_1
iload_2
ldc 113
imul
iadd
istore_1
stop_0:
start_3:
iload_2
iconst_0
if_icmple stop_4
iload_1
iload_3
imul
istore_1
iinc 2 -1
goto start_3
stop_4:
```

Smaller and faster code:

- remove unnecessary operations;
- simplify control structures; and
- replace complex operations by simpler ones (strength reduction).

This is what the JOOS optimizer does.

Later, we shall look at:

- JIT compilers; and
- more powerful optimizations based on static analysis.

Larger, but faster code: tabulation.

The sine function may be computed as:

$$\texttt{sin}(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots...$$

or looked up in a table:

sin(0.0)	0.000000
sin(0.1)	0.099833
sin(0.2)	0.198669
sin(0.3)	0.295520
sin(0.4)	0.389418
sin(0.5)	0.479426
sin(0.6)	0.564642
sin(0.7)	0.644218

Larger, but faster code: loop unrolling.

The loop:

```
for (i=0; i<2*N; i++) {
    a[i] = a[i] + b[i];
}
```

is changed into:

```
for (i=0; i<2*N; i=i+2) {
    j = i+1;
    a[i] = a[i] + b[i];
    a[j] = a[j] + b[j];
}
```

which reduces the overhead and may give a 10–20% speedup.

The optimizer must undo fancy language abstractions:

- variables abstract away from registers, so the optimizer must find an efficient mapping;
- control structures abstract away from gotos, so the optimizer must construct and simplify a goto graph;
- data structures abstract away from memory, so the optimizer must find an efficient layout;

⋮

- method lookups abstract away from procedure calls, so the optimizer must efficiently determine the intended implementations.

Continuing: the OO language BETA unifies as *patterns* the concepts:

- abstract class;
- concrete class;
- method; and
- function.

A (hypothetical) optimizing BETA compiler must attempt to classify the patterns to recover that information.

Example: all patterns are allocated on the heap, but 50% of the patterns are methods that could be allocated on the stack.

Difficult compromises:

- a high abstraction level makes the development time cheaper, but the run-time more expensive; however
- high-level abstractions are also easier to analyze, which gives optimization potential.

Also:

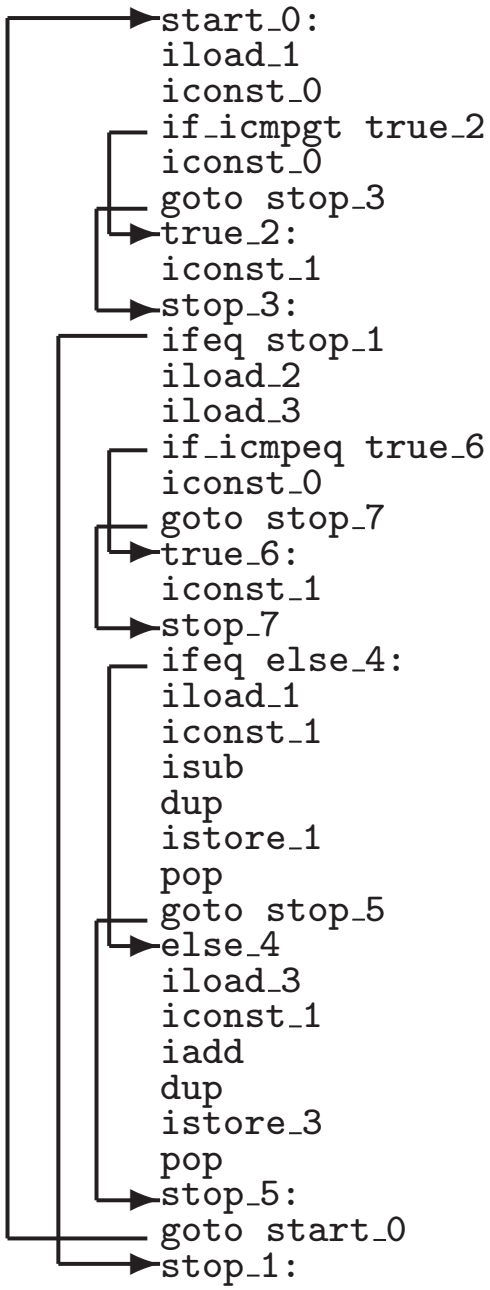
- an optimizing compiler makes run-time more efficient, but compile-time less efficient;
- optimizations for speed and size may conflict; and
- different applications may require different optimizations.

The JOOS peephole optimizer:

- works at the bytecode level;
- looks only at *peepholes*, which are sliding windows on the code sequence;
- uses *patterns* to identify and replace inefficient constructions;
- continues until a global fixed point is reached; and
- optimizes both speed and space.

The optimizer considers the goto graph:

```
while (a>0) {
  if (b==c) a=a-1; else c=c+1;
}
```



To capture the goto graph, the labels for a given code sequence are represented as an array of:

```
typedef struct LABEL {
   char *name;
   int sources;
   struct CODE *position;
} LABEL;
```

where:

- the array index is the label's number;
- the field `name` is the textual part of the label;
- the field `sources` indicates the in-degree of the label; and
- the field `position` points to the location of the label in the code sequence.

Operations on the goto graph:

- inspect a given bytecode;
- find the next bytecode in the sequence;
- find the destination of a label;
- create a new reference to a label;
- drop a reference to a label;
- ask if a label is dead (in-degree 0);
- ask if a label is unique (in-degree 1); and
- replace a sequence of bytecodes by another.

Inspect a given bytecode:

```
int is_istore(CODE *c, int *arg)
{ if (c==NULL) return 0;
  if (c->kind == istoreCK) {
     (*arg) = c->val.istoreC;
     return 1;
  } else {
     return 0;
  }
}
```

Find the next bytecode in the sequence:

```
CODE *next(CODE *c)
{ if (c==NULL) return NULL;
  return c->next;
}
```

Find the destination of a label:

```
CODE *destination(int label)
{ return currentlabels[label].position;
}
```

Create a new reference to a label:

```
int copylabel(int label)
{ currentlabels[label].sources++;
  return label;
}
```

Drop a reference to a label:

```
void droplabel(int label)
{ currentlabels[label].sources--;
}
```

Ask if a label is dead (in-degree 0):

```
int deadlabel(int label)
{ return currentlabels[label].sources==0;
}
```

Ask if a label is unique (in-degree 1):

```
int uniquelabel(int label)
{ return currentlabels[label].sources==1;
}
```

Replace a sequence of bytecodes by another:

```
int replace(CODE **c, int k, CODE *r)
{ CODE *p;
  int i;
  p = *c;
  for (i=0; i<k; i++) p=p->next;
  if (r==NULL) {
     *c = p;
  } else {
     *c = r;
     while (r->next!=NULL) r=r->next;
     r->next = p;
  }
  return 1;
}
```

The expression:

```
x = x + k
```

may be simplified to an increment operation, if $0 \leq \texttt{k} \leq 127$.

Corresponding JOOS peephole pattern:

```
int positive_increment(CODE **c)
{ int x,y,k;
  if (is_iload(*c,&x) &&
      is_ldc_int(next(*c),&k) &&
      is_iadd(next(next(*c))) &&
      is_istore(next(next(next(*c))),&y) &&
      x==y && 0<=k && k<=127) {
     return replace(c,4,makeCODEiinc(x,k,NULL));
  }
  return 0;
}
```

We may attempt to apply this pattern anywhere in the code sequence.

The algebraic rules:

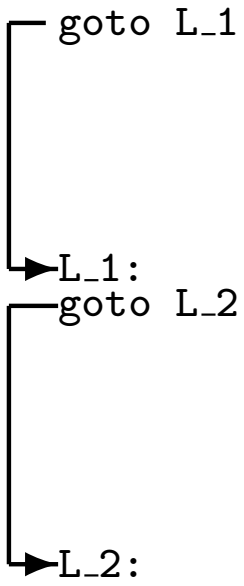
```
x * 0 = 0
x * 1 = x
x * 2 = x + x
```

may be used to simplify some operations.

Corresponding JOOS peephole pattern:

```
int simplify_multiplication_right(CODE **c)
{ int x,k;
  if (is_iload(*c,&x) &&
      is_ldc_int(next(*c),&k) &&
      is_imul(next(next(*c)))) {
     if (k==0)
        return replace(c,3,makeCODEldc_int(0,NULL));
     else if (k==1)
        return replace(c,3,makeCODEiload(x,NULL));
     else if (k==2)
        return replace(c,3,makeCODEiload(x,
                           makeCODEdup(
                           makeCODEiadd(NULL))));
     return 0;
  }
  return 0;
}
```

A part of the goto graph:



may be simplified by short-circuiting the jump to L_1.

Corresponding JOOS peephole pattern:

```
int simplify_goto_goto(CODE **c)
{ int l1,l2;
  if (is_goto(*c,&l1) &&
      is_goto(next(destination(l1)),&l2) && l1>l2) {
     droplabel(l1);
     copylabel(l2);
     return replace(c,1,makeCODEgoto(l2,NULL));
  }
  return 0;
}
```

The JOOS peephole pattern:

```
int simplify_astore(CODE **c)
{ int x;
  if (is_dup(*c) &&
      is_astore(next(*c),&x) &&
      is_pop(next(next(*c)))) {
     return replace(c,3,makeCODEastore(x,NULL));
  }
  return 0;
}
```

is clearly sound, but will it ever be useful?

Yes, the assignment statement:

```
a = b;
```

generates the code:

```
aload_2
dup
astore_1
pop
return
```

because of our simpleminded strategy.

Coding assignments:

```
void codeEXP(EXP *e) {
  case assignK:
    codeEXP(e->val.assignE.right);
    code_dup();
    ......
      case formalSym:
        if (e->val.assignE.leftsym->
               val.formalS->type->kind==refK) {
          code_astore(e->val.assignE.leftsym->
                         val.formalS->offset);
        } else {
          code_istore(e->val.assignE.leftsym->
                         val.formalS->offset);
        }
        break;
```

To avoid the dup, we must know if the assigned value is needed later; this information must then flow back to the code:

```
void codeSTATEMENT(STATEMENT *s) {
  case expK:
    codeEXP(s->val.expS);
    if (s->val.expS->type->kind!=voidK) {
       code_pop();
    }
    break;
```

to decide whether to pop or not. A peephole pattern is simpler and more modular.

Any collection of peephole patterns:

```
typedef int(*OPTI)(CODE **);

#define MAX_PATTERNS 100

int init_patterns() {
  ADD_PATTERN(simplify_multiplication_right);
  ADD_PATTERN(simplify_astore);
  ADD_PATTERN(positive_increment);
  ADD_PATTERN(simplify_goto_goto);
  return 1;
}
```

can be applied to a goto graph in a fixed point process:

```
repeat
    for each bytecode in succession do
        for each peephole pattern in succession do
            repeat
                apply the peephole pattern
                to the bytecode
            until the goto graph didn't change
        end
    end
until the goto graph didn't change
```

JOOS code for the fixed point driver:

```
int optiCHANGE;

void optiCODEtraverse(CODE **c)
{ int i,change;
  change = 1;
  if (*c!=NULL) {
     while (change) {
       change = 0;
       for (i=0; i<OPTS; i++) {
           change = change | optimization[i](c);
       }
       optiCHANGE = optiCHANGE || change;
     }
     if (*c!=NULL) optiCODEtraverse(&((*c)->next));
  }
}

void optiCODE(CODE **c)
{ optiCHANGE = 1;
  while (optiCHANGE) {
    optiCHANGE = 0;
    optiCODEtraverse(c);
  }
}
```

Why does this process terminate?

Each peephole pattern does not necessarily make the code smaller.

To demonstrate termination for our examples, we use the lexicographically ordered measure:

$$< \#\text{bytecodes}, \#\texttt{imul}, \sum_{\texttt{L}} |\mathit{gotochain}(\texttt{L})| >$$

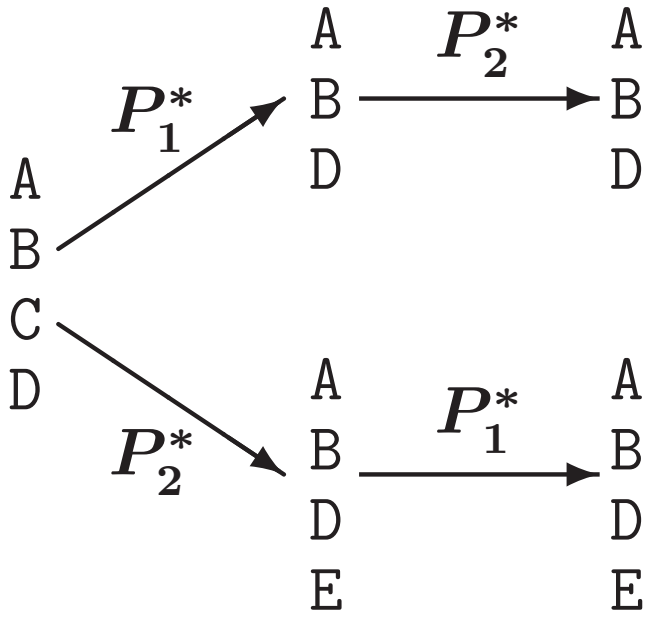
which can be seen to become strictly smaller after each application of a peephole pattern.

The goto graph obtained as a fixed point is *not* unique.

It depends on the sequence in which the peephole patterns are applied.

That does not happen for the four examples given, but consider the two peephole patterns:

They clearly do not commute:



The effect of the JOOS A+ optimizer (using 40 peephole patterns):

Program	joosa+	joosa+ -O
AllComponents	907	861
AllEvents	1056	683
Animator	184	180
Animator2	568	456
ConsumeInteger	164	107
DemoFont	97	89
DemoFont2	213	147
DrawArcs	60	60
DrawPoly	94	90
Imagemap	470	361
MultiLineLabel	526	406
ProduceInteger	149	96
Rectangle2	58	58
ScrollableScribble	566	481
ShowColors	88	68
TicTacToe	1471	1211
YesNoDialog	315	248

The word "optimizer" is somewhat misleading, since the code is not optimal but merely better.

Suppose $OPM(\boldsymbol{G})$ is the shortest goto graph equivalent to $\boldsymbol{G}$.

Clearly, the shortest diverging goto graph is:

$$\boldsymbol{D}_{\text{min}} \quad = \quad \begin{array}{l} \texttt{L:} \\ \texttt{goto L} \end{array}$$

We can then decide the Halting problem on an arbitrary goto graph $\boldsymbol{G}$ as:

$$OPM(\boldsymbol{G}) = \boldsymbol{D}_{\text{min}}$$

Hence, the program OPM cannot exist.

The testing strategy for the optimizer has three phases.

First a careful argumentation that each peephole pattern is sound.

Second a demonstration that each peephole pattern is realized correctly.

Third a statistical analysis showing that the optimizer improves the generated programs.