Optimization

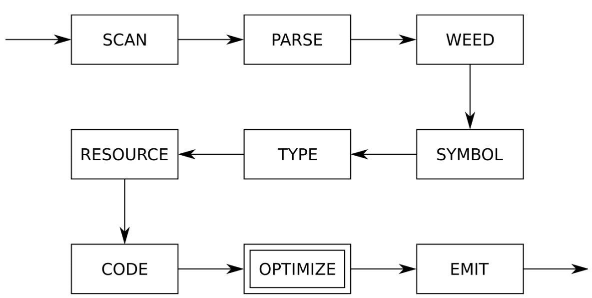
COMP 520: Compiler Design (4 credits)

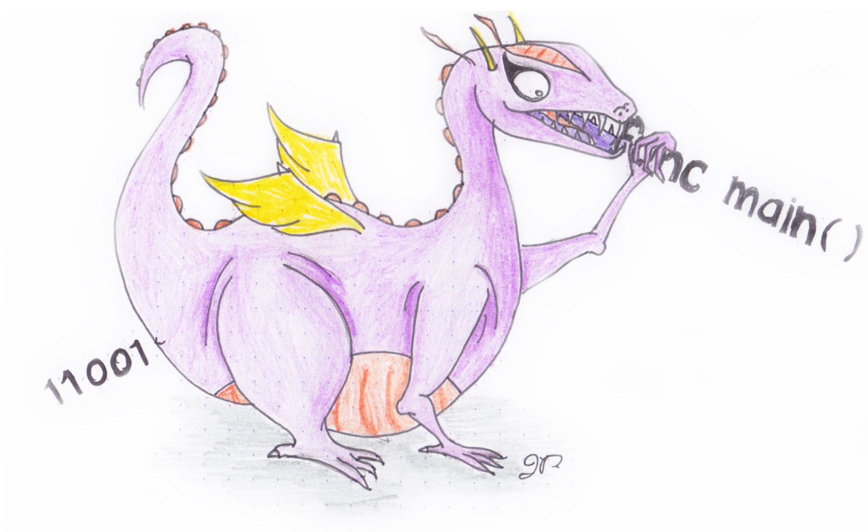
Alexander Krolik

alexander.krolik@mail.mcgill.ca

MWF 10:30-11:30, TR 1100

http://www.cs.mcgill.ca/~cs520/2020/





Dot Gitignore

Announcements (Wednesday, February 19th)

Milestone 1

- Any questions?
  - Weeding cases, what are they?
  - Semicolon insertion rule
- **Due:** Saturday, February 22nd 11:59 PM

Midterm

- **Date:** Tuesday, February 25th 6:00 - 7:30 PM in RPHYS 112
- **Review:** Monday, February 24th in class
- Sample midterm from 2019

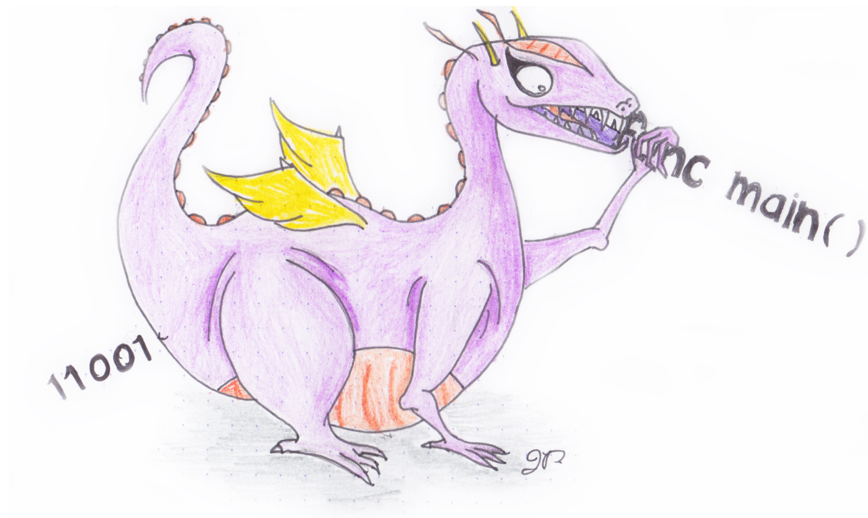
Optimization

Introduction

Peephole

Contest

Thought



Optimization

We typically think of optimization in terms of speed, but an *optimizer* can focus on any of:

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

Ideally

The best optimizations achieve all goals – but this is difficult to accomplish in general. These goals often conflict, since a larger program may in fact be faster.

- Loop unrolling;
- Type/shape specialization;
- etc.

Optimizations for Space

Optimizations for *space* reduce code size by replacing sequences of instructions with a smaller set.

Over time

- 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 was small and expensive, so Java compilers optimized for space; but
- Today Internet bandwidth is larger and cheaper, so we optimize for speed again.

$\Rightarrow$ Optimizations are driven by economy!

Optimizations for Speed

Optimizations for *speed* improve the execution performance of the program.

Over time

- Historically very important to gain acceptance for high-level languages; and
- Are still important, since the software always strains the limits of the hardware.

These types of optimizations form the bulk of modern optimizing compilers.

Difficulty

Optimizations for speed are a battle, mapping the programming language to the hardware

- Challenged by ever higher abstractions in programming languages; and
- Must constantly adapt to changing microprocessor architectures.

Optimizations for Speed

Regardless of the language and underlying hardware, several common optimization areas include (from low-level to high-level)

- Cache performance;
- Parallel/vectorization;
- Loop invariants;
- Common-subexpression elimination (CSE)/dead code removal; and
- $\cdots$

Optimization Passes

Optimizations may take place at various levels of program transformation/execution

- At the source code level (programmer);
- 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.

Optimization Strategy

Choosing an optimization strategy is a balance between the needs of the programmer/user

- Writing time (programmer);
- Compilation time (programmer or user);
- Execution time (user).

Compiler pipeline

We must decide the most effective phase to perform each optimization depending on

- Necessary information/representations;
  - Machine characteristics (low-level);
  - Programming language constructs (high-level);
- Runtime vs offline; and more

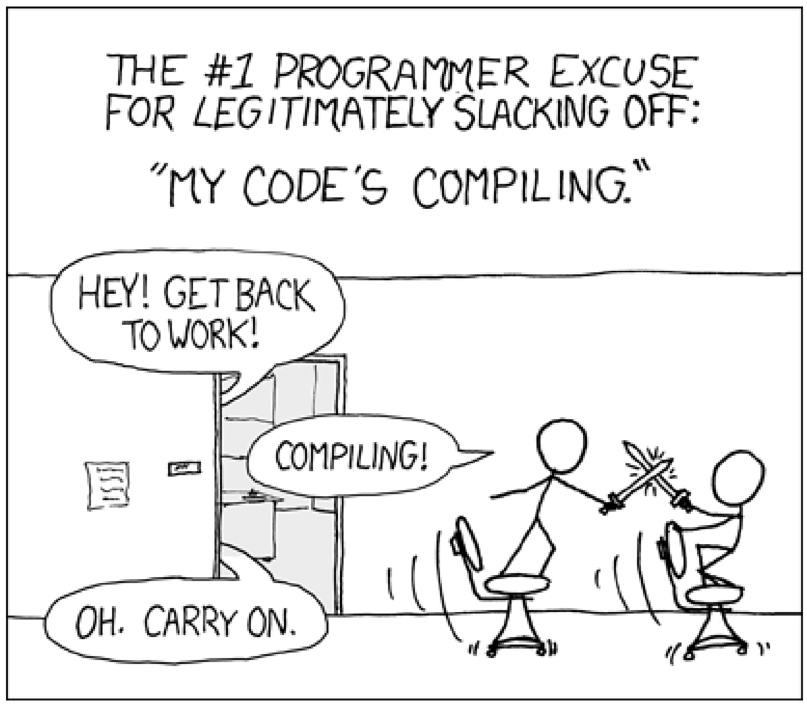
Note: The "best" strategy is still very much up for debate.

Optimization Considerations

The following slides outline several considerations we commonly see in compiler design

1. Programmer vs. compiler;
2. Size vs. speed;
3. Abstraction vs. low-level.

Of course, there are many many more!



http://xkcd.com/303/

Optimization from a Programmer's Perspective

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?

Optimization from a Programmer's Perspective

If you said `LOOP #2` . . . you were (mostly) 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

Optimization from a Programmer's Perspective

If you said `LOOP #2` . . . you were (mostly) 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

- Hand-optimization does improve performance with the optimizer off, but not with it on!
- Pointers confuse most C compilers! Keeping array structures is much easier to optimize.
- In general, write clear C code; it is easier for both the programmer and compiler to understand.

Optimization: Smaller and Faster

Intuitively, reducing the number of instructions to execute can improve program performance

- Remove unnecessary operations;
- Simplify control structures; and
- Replace complex operations by simpler ones (strength reduction).

This is what the JOOS peephole optimizer does.

Optimization: Smaller and Slower

On the other hand, reducing the code size (or keeping it small) might not improve the performance

- Function calls instead of inlining is costly;
- Not unrolling loops leads to more jumps;
- CSE (common-subexpression elimination) may increasing register pressure.

Conclusion

Even though the JOOS optimizer targets size for speed, it is important to not always equate improvements in size with improvements in speed.

Optimization: Larger and Faster (Tabulation)

In some instances, expanding the code size can improve performance. Tabulation is one such approach which replaces function calls with an approximation

Sine function

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

Optimization using a lookup 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

Optimization: Larger and Faster (Loop Unrolling)

Loop unrolling reduces the overhead of jumping and condition testing by merging adjacent iterations. Given a loop bound multiple of two

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

We can rewrite the code by merging pairs of iterations (unroll factor 2)

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

Loop unrolling can give a 10–20% speedup. What is a potential disadvantage? How does this work for loop bounds that may not be a multiple of the unroll factor?

Aside: Duff's Device

Handles loop unrolling where the loop bound may not be a multiple of the unroll factor

```
do {
   *to = *from++;
} while(--count > 0);
```

We can unroll with a factor of 8 to produce the following code, which assumes the loop bound is a multiple of 8

```
register n = count / 8;
do {
   *to = *from++;
   *to = *from++;
   *to = *from++;
   *to = *from++;
   *to = *from++;
   *to = *from++;
   *to = *from++;
   *to = *from++;
} while (--n > 0);
```

Aside: Duff's Device

To handle the case where the loop bound is not a multiple of 8, we use the following quirky C code, where we "jump" into the unrolled loop using a switch statement

```
register n = (count + 7) / 8;
switch (count % 8) {
   case 0: do { *to = *from++;
   case 7:      *to = *from++;
   case 6:      *to = *from++;
   case 5:      *to = *from++;
   case 4:      *to = *from++;
   case 3:      *to = *from++;
   case 2:      *to = *from++;
   case 1:      *to = *from++;
   } while (--n > 0);
}
```

For those interested: `https://en.wikipedia.org/wiki/Duff%27s_device`

Optimizing High-Level Languages

High-level languages provide fancy language abstractions that are unrelated to the underlying hardware. The optimizer must therefore undo these abstractions for execution

- 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.

Difficult compromises

- A high abstraction level makes the development time cheaper, but can make the run-time more expensive as they need to be mapped to hardware; however
- High-level abstractions are also easier to analyze, which gives optimization potential.

Optimizing High-Level Languages

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.

Other Optimizations

These are but a fraction of optimization avenues. Later, we will look at

- Parallelism through GPUs;
- JIT compilers (high level); and
- More powerful optimizations based on static analysis (COMP 621).

But there are many, many more.

Optimization considerations

- An optimizing compiler makes run-time more efficient, but compile-time less efficient; and
- Different applications may require different optimizations.

Optimization Takeaways

As a programmer, you should have the following in mind whenever you write your programs.

1. Trust your compiler;
   - Use high-level language features that can be easily optimized, and avoid low-level features that may confuse compilers;
2. Speed and size are not necessarily related, and often conflict;
   - Increasing/decreasing size can both improve/hurt speed;
3. High-level languages require extensive optimization effort;
   - Abstraction is great for the programmer, hard for the compiler;
4. Optimization is a complex problem, and you are likely never done.

Announcements (Friday, February 21st)

Milestone 1

- Any last minute questions?
- **Due:** Saturday, February 22nd 11:59 PM

Midterm

- **Review:** Monday, February 24th in class
- **Date:** Tuesday, February 25th 6:00 - 7:30 PM in RPHYS 112
- Class Wednesday, February 26th cancelled
- Sample midterm from 2019

Milestones

- Peephole out today! **Due:** Friday, April 10th 11:59 PM

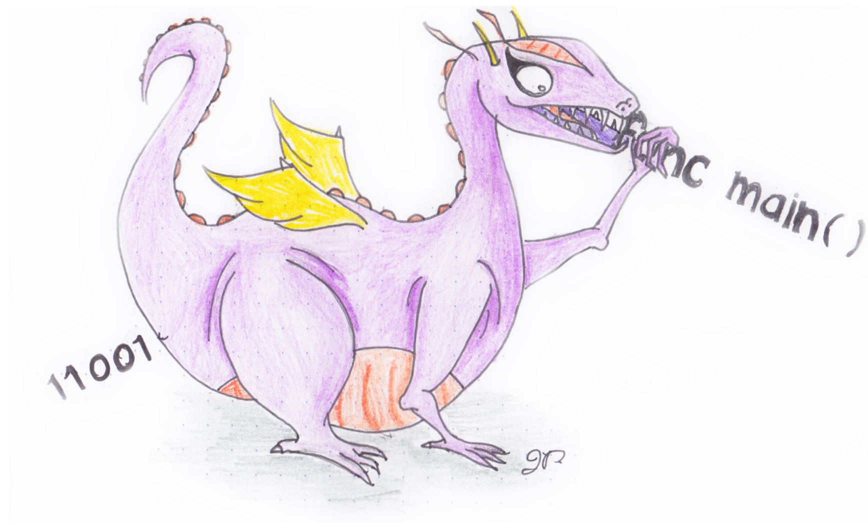
Optimization

Introduction

Peephole

Contest

Thought



Peephole Optimizer

In this class we will focus on a simple type of optimization (more detailed optimizations are discussed in COMP 621)

- 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.

Example

Remove unnecessary dup/pop (generated from assignments)

```
dup
istore_{x}      ⟹      istore_{x}
pop
```

JOOS Optimization

```
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
  ...
```

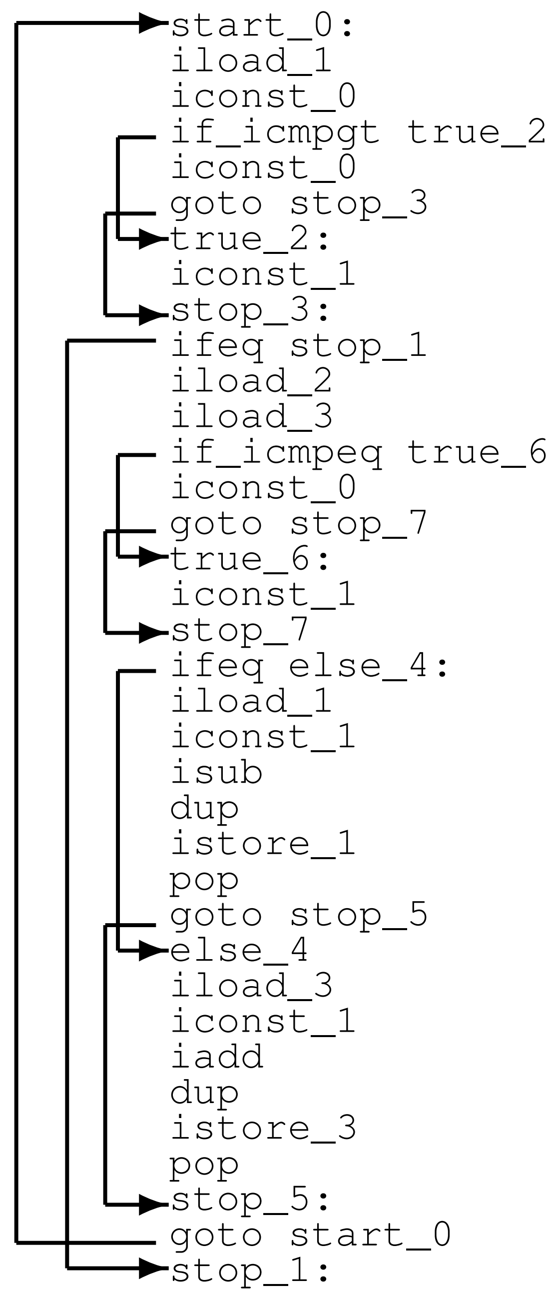
```
  iload_1
  iload_2
  imul
  iload_3
  iadd
  istore_3
  iload_3
  iload_1
  if_icmpge stop_0
  iload_1
  iload_2
  ldc 113
  imul
  iadd
  istore_1
stop_0:
start_3:
  iload_2
  ifle stop_4
  iload_1
  iload_3
  imul
  istore_1
  iinc 2 -1
  goto start_3
stop_4:
```

Optimizer Goto Graph

To optimize, we can't simply assume instructions are given in the order they are executed.

Instead, the optimizer works on a structure called a *goto graph* that represents the jumps in a program.

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



Optimizer Goto Graph

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

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

Defined as

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

Operations on the Goto Graph

The optimizer acts on the goto graph and may

- Inspect a given bytecode (get the instruction kind);
- 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**.

Optimizer - Instructions

A peephole optimizer replaces one sequence of instructions by another using patterns.

- Check each instruction is in the pattern (`is_<inst>`); and
- Traverse the bytecode sequence (`next`).

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;
   }
}
```

Note you can also return instruction arguments using the pointer `arg`.

Find the next bytecode in the sequence

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

Optimizer - Labels

Optimizations may also traverse the goto graph and evaluate jump targets.

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--;
}
```

The latter 2 operations are used when applying peephole transformations.

Optimizer - Labels

Optimizations may check properties of labels (for instance to remove dead labels).

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;
}
```

Optimization - Replace

A peephole pattern identifies a sequence of bytecode to optimize, and replace it by another.

```
int replace(CODE **c, int k, CODE *r) {
   CODE *p = *c;
   for (int 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;
}
```

1. Find the first instruction that is not replaced (`i`);
2. Insert the new sequence (if there is one); and
3. Attach the end of the new sequence to instruction `i`.

Peephole Pattern - Positive Increment

An increment to a local variable may be simplified to an increment operation, if $0 \leq \mathtt{k} \leq 127$

```
x = x + k
```

Peephole pattern

For this pattern, not only do we have a transformation, but also a restriction on the argument

```
iload_{x}
ldc_int {k}        ⟹      iinc {x} {k} // 0 <= k <= 127
iadd
istore_{x}
```

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;
}
```

Peephole Pattern - Algebraic Rules

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

Peephole pattern (# 1)

```
iload_{x}
lconst_0        ⟹        iconst_0
imul
```

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;
}
```

Peephole Pattern - Goto Goto

A part of the goto graph may be simplified by short-circuiting the jump to `L_1`

```
goto L_1



L_1:
goto L_2



L_2:
```

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(l12, NULL));
   }
   return 0;
}
```

Peephole Pattern - Goto Goto

Why the condition `l1 > l2`?

```
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;
}
```

Consider the following bytecode

```
l1: goto l1
```

What will happen without this condition?

Peephole Pattern - Simplify astore

The following JOOS peephole pattern removes an unnecessary `dup`/`pop` pair of instructions

```
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;
}
```

It is clearly sound, but will it ever be useful?

Peephole Pattern - Simplfy astore

Yes! Consider the following expression statement:

```
a = b;
```

We generate the assignment expression without the surrounding statement context - and therefore leave the value on the top of the stack.

```
aload_2
dup
astore_1
pop
```

Recall, the final pop instruction is generated at the statement level.

Peephole Pattern - Simplify astore

The context agnostic generation for assignment expressions inserts the `dup` instruction by default

Corresponding JOOS source code

```
void codeEXP(EXP *e) {
    case assignK:
        codeEXP(e->val.assignE.right);
        code_dup();
        switch (e->val.assignE.leftsym->kind) {
            [...]
            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;
```

This handles chains of assignments `a = b = c` where the value is later needed.

Peephole Pattern - Simplify astore

To avoid the `dup` in the assign template

- We must know if the assigned value is needed later (contextual information); and
- It must also flow the decision back to the enclosing code below.

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

A peephole pattern is simpler and more modular.

Peephole Optimization

The peephole optimizer applies the collection of patterns 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
```

Peephole Optimization Termination

Why does this process terminate?

- Each peephole pattern does not necessarily make the code smaller; so
- To demonstrate termination for our examples, we use the lexicographically ordered measure

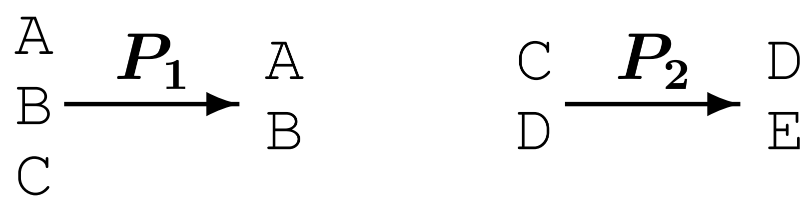
$$< \#\text{bytecodes}, \#\texttt{imul}, \sum_{\texttt{L}} |\textit{gotochain}(\texttt{L})| >$$

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

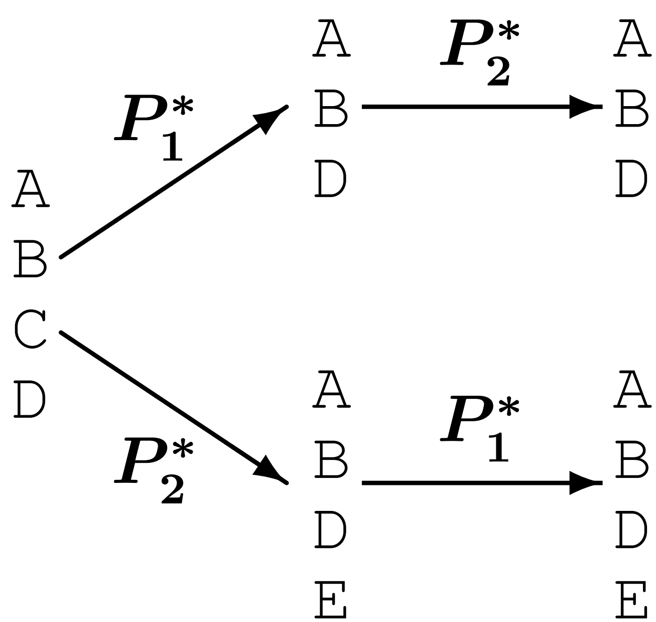
Peephole Optimization Fixed Point

- The goto graph obtained as a fixed point is *not* unique; since
- 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:



These patterns do not commute



"Optimizer"

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

Can we find the optimal?

Suppose *OPM*(G) is the shortest goto graph equivalent to G. The shortest diverging goto graph is

D_{min} =

```
L:
goto L
```

We can then decide the Halting problem on an arbitrary goto graph G as

$$\mathit{OPM}(G) = D_{\text{min}}$$

Hence, the program *OPM* cannot exist.

Testing

The testing strategy for the optimizer has three phases:

1. A careful argumentation that each peephole pattern is sound;
    - Local variables have the same values;
    - Stack height changes by the same amount;
    - All paths yield the same outcome;
2. A demonstration that each peephole pattern is realized correctly; and
3. A statistical analysis showing that the optimizer improves the generated programs.

Optimization

Introduction

Peephole

Contest

Thought

JOOS Peephole Optimizer (patterns.h)

```
/* patterns here */

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;
}

[...]

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

JOOS Peephole Optimizer (Fixed Point Driver)

```
int optiCHANGE;

void optiCODEtraverse(CODE **c) {
   int change = 1;
   if (*c != NULL) {
      while (change) {
         change = 0;
         for (int 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);
   }
}
```

JOOS A+ Peephole Optimizer (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

Peephole Competition

The peephole assignment is a yearly competition to see who can achieve the highest reduction in the size of JVM bytecode.

- Start with the A- JOOS compiler (`https://github.com/comp520/Peephole-Template`);
- Add patterns that reduce the size of the code;
- Compete against your fellow classmates!
- *Work in your GoLite project teams*

Results from previous years

- **A-**: -1.4%
- **A+**: -21.9%
- **2017**: -21.8%
- **2018**: -31.9%
- **2019**: -30.4%

Peephole Competition

Requirements

For each pattern that you add to `patterns.h` you must:

1. Ensure that it is sound;
2. Check a fixed point will be reached; and
3. Put a comment which clearly describes the pattern.

Workflow

As you work on your submission, your workflow will likely be as follows:

1. Generate the bytecode for all benchmarks (`count.sh` script)
2. Analyze the generated bytecode (`.j` files) for inefficiencies
3. Design a pattern that improves the code size
4. Test for (a) soundness; and (b) code size improvement

The final evaluation is on a set of public and hidden benchmarks.

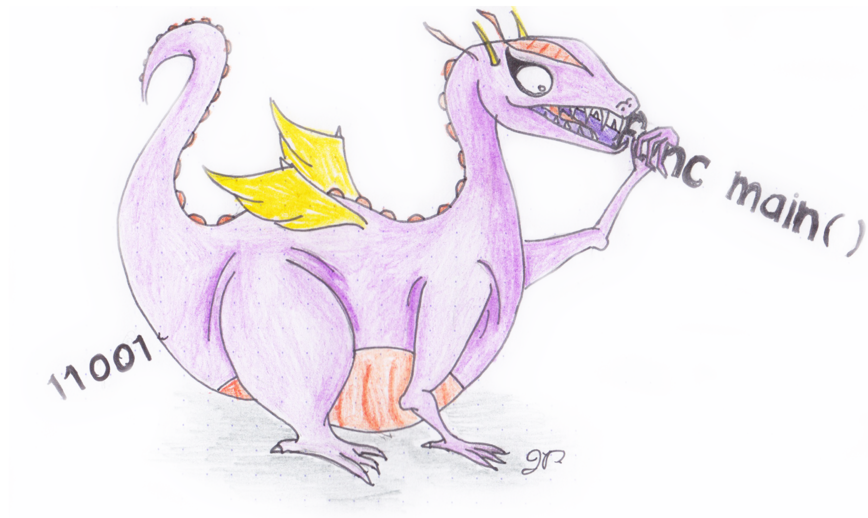
Optimization

Introduction

Peephole

Contest

Thought



There is a fine line between "optimization" and "not being stupid"

- R. Kent Dybvig