

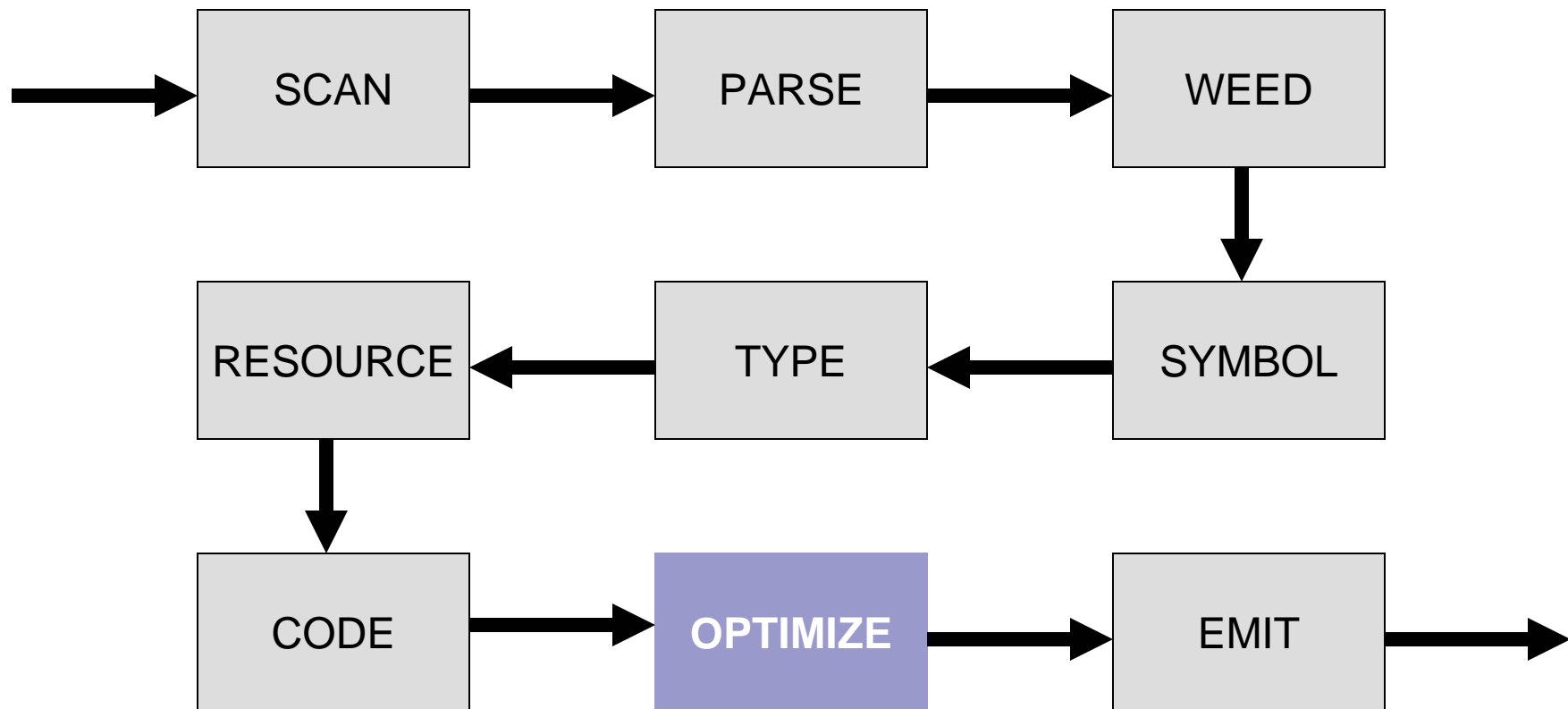


Compiler

Bytecode Optimization

© Copyright 2005, Matthew B. Dwyer and Robby. The syllabus and lectures for this course are copyrighted materials and may not be used in other course settings outside University of Nebraska-Lincoln and Kansas State University in their current form or modified form without express written permission of one of the copyright holders. During this course, students are prohibited from selling notes to or being paid for taking notes by any person or commercial firm without the express written permission of one of the copyright holders.

Compiler Architecture





Optimizer

- Focuses on

- ☐ Reducing run-time cost
- ☐ Reducing code size
- ☐ Reducing power consumption

- Goals often conflict

- ☐ Usually cannot find an optimal solution
- ☐ Code *improver* is a more accurate term



Optimizing Space

Optimizations are driven by economics!

- historically very important, because memory was small and expensive;
- when memory became large and cheap, optimizing compilers traded space for speed; but
- now Internet bandwidth is small and expensive, so Java compilers optimize for space.



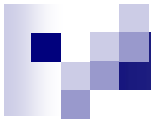
Optimizing Time

- historically very important to gain acceptance for high-level languages;
- still important, since software features are constantly expanding to strain the limits of hardware;
- are challenged by ever higher level abstractions in language;
- must constantly adapt to changing micro-processor architectures.



Optimizing Power

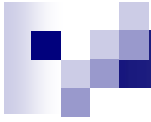
- historically these weren't an issue, since computers were plugged into the wall;
- mobile devices like laptops, PDAs and cell-phones are full-blown computing platforms;
- power consumption is the dominant issue in compiling for mobile platforms today.



Where to Optimize?

- Source code level
- Intermediate representation (e.g., AST)
- Virtual machine code (e.g., JVM)
- Binary machine code level
- During run-time (e.g., JIT compilers)

Aggressive optimization requires lots of small contributions at all levels.



Bytecode Optimizer

Remove unnecessary operations

- especially those at the boundaries of templates

Simplify control structures

Replace complex operators by simpler ones

Later on we will look at:

- JIT compilers
- More powerful “global” optimizations



Tabulation

- Sometimes we can trade space for time
- Consider the sine function computed as:

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

- Alternatively we could use:

| | |
|-----------------------|----------|
| <code>sin(0.0)</code> | 0.000000 |
| <code>sin(0.1)</code> | 0.099833 |
| <code>sin(0.2)</code> | 0.198669 |
| <code>sin(0.3)</code> | 0.295520 |
| <code>sin(0.4)</code> | 0.389418 |
| <code>sin(0.5)</code> | 0.479426 |
| <code>sin(0.6)</code> | 0.564642 |
| <code>sin(0.7)</code> | 0.644218 |
| | |



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 to 20% speedup.



Issues

- The optimizer must undo fancy language abstractions, e.g.:
 - 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.



Compromises

- a high abstraction level makes the development time cheaper, but the run-time more expensive;
- 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.



Peephole Bytecode 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.



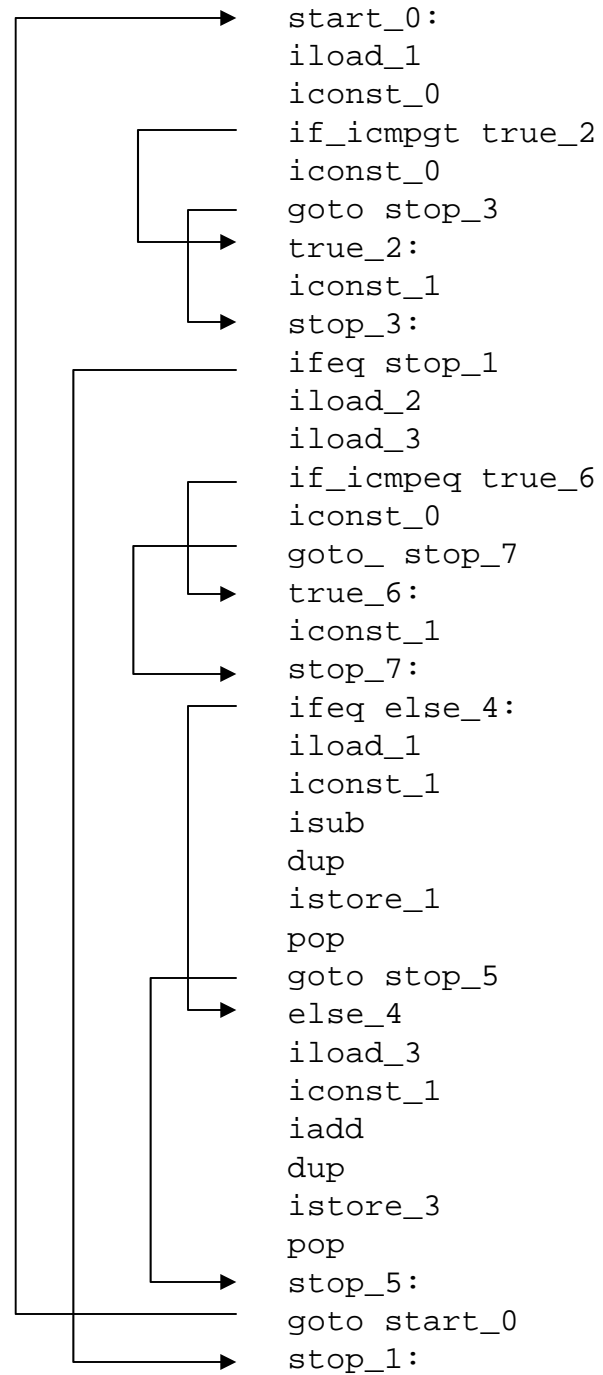
Peephole Bytecode Optimizer

- Uses a program representation called the *goto graph*

- Explicitly connects branches to labels

- For example

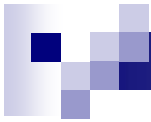
```
while (a > 0) {  
    if (b == c) { a = a - 1; } else { c = c + 1; }  
}
```





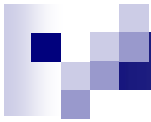
Goto Graph

- Represent labels for a code block as an array of a label structure consisting
 - the name of the label
 - the in-degree of the label
 - the bytecode position of the label



Goto Graph Operations

- 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.



Generic Peephole Pattern

- Takes a sequence of bytecodes
- Tests a condition on some part of that sequence, e.g., the first few bytecodes
- If test passes, implements a transformation of the bytecodes
- Returns 1 if transformation is made and 0 otherwise

Increment Identification

The expression: $x = x + k$ may be simplified to an increment operation, if $0 \leq k \leq 255$

ILOAD *i*
BIPUSH *k*
IADD
ISTORE *i*



IINC *i k*

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

Algebraic Simplifications

Exploit arithmetic properties


$$x * 0 = 0$$

$$x * 1 = x$$

$$x * 2 = x + x$$


Example Patterns

ILOAD *i*
ICONST_0
IMUL




ICONST_0

ILOAD *i*
ICONST_1
IMUL



ILOAD *i*

ILOAD *i*
ICONST_2
IMUL




ILOAD *i*
DUP
IADD

Jump Optimizations

Short-circuit chains of jumps, e.g.,

```
goto L1      goto L7
...
L1:
goto L7
...
L7:
```



For You To Do

- Is the following pattern useful?

DUP
ASTORE *i*
POP



ASTORE *i*



Java Statement Expressions

The assignment statement:

`a = b;`

generates the code:

`ALOAD 2`

~~`DUP`~~

`ASTORE 1`

~~`POP`~~

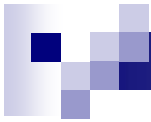
because of our simpleminded strategy.



Assignments

To avoid the `dup`, we must know if the assigned value is needed later; this information must then flow back to the code gen for the expression.

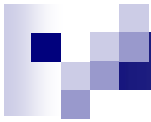
To decide whether to `pop` or not. A peephole pattern is simpler and more modular.



Peephole Optimizer Engine

Any collection of peephole patterns 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 doesn't change
    end
  end
end
until the goto graph doesn't change
```



For You To Do

- Does this process terminate?
- Does it depend on the peephole patterns?
- If so, what characteristics should the peephole patterns have to ensure termination?
- If not, what characteristic of the driver assures termination?



Monotonicity

- Termination is a property of the monotonicity of the peephole pattern transformations
 - i.e., they always make some *measure* smaller
- Some patterns do not make the code shorter, e.g., jump chaining
- What is the measure?



Lexicographic Measure

- The measure has three components
 - Total number of bytecodes
 - Number of `IMUL` bytecodes
 - Sum of the length of all goto chains
- Seems kind of cooked up to account for the *work* that our patterns do in simplifying the program
 - Nature of measure is unimportant, it's existence is all that matters
 - If you define new patterns make sure the are monotone with respect to this or an extended measure

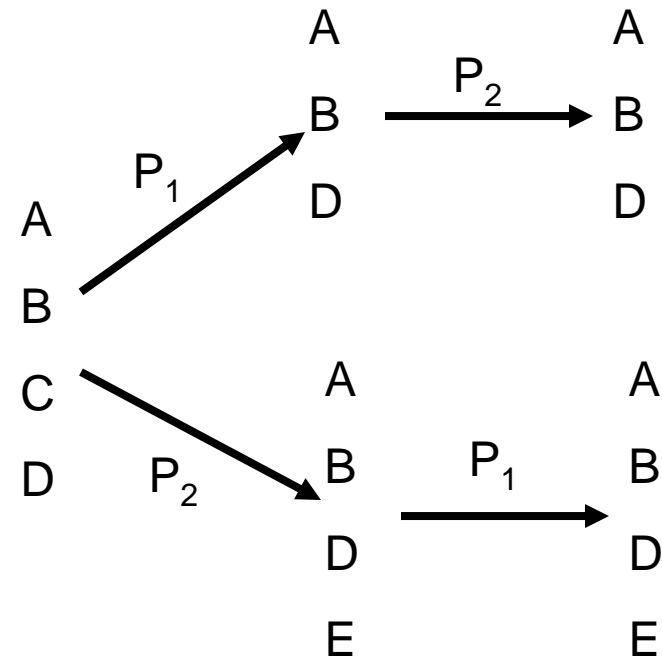
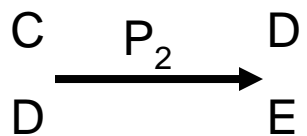
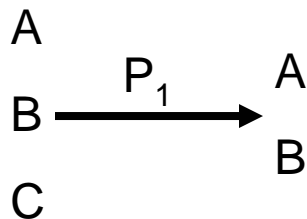


For You to Do

- Does the order of pattern application affect the resultant program?
- Note that the patterns are applied in the order in which you define them in the `optimizations` array

Order Matters (potentially)

- For the example patterns, order doesn't matter, but consider the following patterns:





“Optimizer”

- Suppose $OPM(G)$ is the shortest goto graph equivalent to G .
- The shortest diverging goto graph, D , is:

L:

goto L

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

$$OPM(G) = D$$

hence, the program OPM cannot exist.



Testing the Optimizer

- Carefully argue that each peephole pattern is sound, i.e., it preserves the semantics of the original bytecodes
- Demonstrate that each peephole pattern is coded correctly
- Show statistics that illustrate the improvements possible using your optimizer.