Introduction to Optimizing Compilers
Why Compilers?

Plenty of good reasons

- arguably the most important software out there
  - affect (almost) all existing software
  - improves software productivity
  - helps design and evaluate computer architectures
  - tool to develop other programming tools

- an excellent software engineering study
  - neither too simple, nor too complex (well…)
  - where theory meets practice
    - must be correct in all cases
    - many problems are NP-complete, need to formulate right abstraction
Why Compilers?

- Where theory meets practice: use of mathematical abstraction

There are programmers, and there are tool builders.
Do We Need Optimizing Compilers?

- Demo
  - Fibonacci
    - SnuPL
    - C with gcc and icc (Intel C Compiler)
  - Mandelbrot
    - C with gcc and icc (Intel C Compiler)
The Structure of a Compiler
The Big Picture

```
#include <stdio.h>
#define N 1000

int main() {
    int A[N];
    a[i] = a[i] + 1;
    printf("%d", a[i]);
}
```

```
int printf(...);
...
int main() {
    int A[1000];
    a[i] = a[i] + 1;
    printf("%d", a[i]);
}
```

```
# include <stdio.h>
# define N 1000

int main() {
    int A[N];
    a[i] = a[i] + 1;
    printf("%d", a[i]);
}
```

```
movl $a, %eax
addl (%eax, %ebx, 4), %ecx
...
call printf
```

```
6: e8 fc ff ff ff  call 7<main+0x14>
7: R_386_PC32 printf
```

```
80483ba: e8 09 00 00 00  call 80483c8
```
Typical Phases of a Compiler

- Character stream
  - Lexical Analyzer
    - Token stream
    - Syntax Analyzer
      - Syntax tree
      - Semantic Analyzer
        - Syntax tree
        - Intermediate Code Generator
  - Symbol Table
    - Intermediate representation (IR)
      - Machine-Independent Code Optimizer
        - Intermediate representation
          - Code Generator
            - Target machine code
              - Machine-Dependent Code Optimizer
                - Target machine code
Optimizations in a Modern Compiler

Scalar replacement of array references
Data-cache optimizations

Procedure integration
Tail-call optimization, including
tail-recursion elimination
Scalar replacement of aggregates
Sparse conditional constant propagation
Interprocedural constant propagation

Global value numbering
Local and global copy propagation
Sparse conditional constant propagation
Dead-code elimination

High-level IR

Partial-redundancy elimination

Low-level IR

Constant folding
Algebraic simplifications including reassociation

Medium-level IR

Local and global common-subexpression elimination
Loop-invariant code motion

High-level IR

Dead-code elimination
Code hoisting
Induction-variable strength reduction
Linear-function test replacement
Induction-variable removal
Unnecessary bounds-checking elimination
Control-flow optimizations

Interprocedural register allocation
Aggregation of global references
Interprocedural I-cache optimizations

In-line expansion
Leaf-routine optimization
Shrink wrapping
Machine idioms
Tail merging
Branch optimizations and conditional moves
Dead-code elimination
Software pipelining with loop unrolling,
variable expansion, register renaming,
and hierarchical reduction
Basic-block and branch scheduling
Register allocation by graph coloring
Basic-block and branch scheduling
Interprocedural I-cache optimization
Instruction prefetching
Data prefetching
Branch prediction

Medium-level IR

Interprocedural I-cache optimizations

Low-level IR

In-line expansion
Leaf-routine optimization
Shrink wrapping
Machine idioms
Tail merging
Branch optimizations and conditional moves
Dead-code elimination
Software pipelining with loop unrolling,
variable expansion, register renaming,
and hierarchical reduction
Basic-block and branch scheduling
Register allocation by graph coloring
Basic-block and branch scheduling
Interprocedural I-cache optimization
Instruction prefetching
Data prefetching
Branch prediction

Link time

data source: S. Muchnick, Advanced Compiler Design & Implementation, Morgan-Kaufmann
Optimizations in a Modern Compiler

- Scalar replacement of array references
- Data-cache optimizations
- Procedure integration
  - Tail-call optimization, including tail-recursion elimination
  - Scalar replacement of aggregates
  - Sparse conditional constant propagation
  - Interprocedural constant propagation
- Sparse conditional constant propagation
  - Global value numbering
    - Local and global copy propagation
      - Sparse conditional constant propagation
      - Dead-code elimination
  - Local and global common-subexpression elimination
    - Loop-invariant code motion
- Partial-redundancy elimination
- Constant folding
- Algebraic simplifications including reassociation
- Dead-code elimination
  - Code hoisting
  - Induction-variable strength reduction
  - Linear-function test replacement
  - Induction-variable removal
  - Unnecessary bounds-checking elimination
  - Control-flow optimizations
- In-line expansion
- Leaf-routine optimization
- Shrink wrapping
- Machine idioms
- Tail merging
- Branch optimizations and conditional moves
- Dead-code elimination
- Software pipelining with loop unrolling, variable expansion, register renaming, and hierarchical reduction
- Basic-block and branch scheduling
- Register allocation by graph coloring
  - Basic-block and branch scheduling
  - Interprocedural l-cache optimization
  - Instruction prefetching
  - Data prefetching
  - Branch prediction
- Interprocedural register allocation
  - Aggregation of global references
  - Interprocedural l-cache optimizations

Data source: S. Muchnick, Advanced Compiler Design & Implementation, Morgan-Kaufmann
Next Class

- Discuss term project

- Till then
  - try to find a team member (use eTL’s message board)
  - get your hands on a copy of the Dragon book