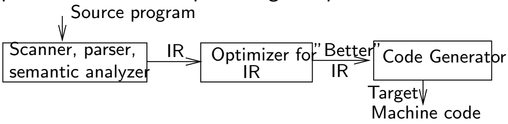
Code Optimization

- Goals of code optimization: remove redundant code without changing the meaning of program.
  Objective:
  1. Reduce execution speed
  2. Reduce code size

  Achieved through code transformation while preserving semantics.
- A very hard problem + non-undecidable, i.e., an optimal program cannot be found in most general case.
- Many complex optimization techniques exist.
  Trade offs: Effort of implementing a technique + time taken during compilation vs. optimization achieved.
  For instance, lexical/semantic/code generation phases require linear time in terms of size of programs, whereas certain optimization techniques may require quadratic or cubic order.
  In many cases simple techniques work well enough.
- Issues:
  - What are principal sources of optimization?
  - When are these optimizations applied?

Classification of optimizations

- Two kinds of classifications of various optimizations:
  - **Time of application**: During which phase of compilation process is an optimization applied?
  - **Scope of application**: What is the area over which optimizations applied? (Local, global, inter-procedural?)
- Time of application: An ideal optimizing compiler structure:



  - Source language optimizations: target independent.
  - Intermediate code generation: majority of machine independent optimizations performed here.
  - Final code generation: (Machine dependent optimizations)
  - Interaction between various phases of optimization: one phase may facilitate other phase. So order of application important.

```
Original code              Transformed code
x = 1;                      x = 1;
...                         ...
y = 0;                      y = 0;
...                         ...
if (y) x = 0;
...                         ...
if (x) y = 1                if (x) y = 1;
```

Code optimization during various phases

- Source language optimizations:
  - Exploit constant bounds in loops and arrays
  - Inhibit code generation of unreachable code segments
  - Unroll loop bodies into equivalent sequential code:

```
Original code               Transformed code
for i in 1..10 loop         A[1][1] := 2;
   a[i][i] = 2*i;           A[2][2] := 4;
end loop                    ...
```

  - Suppress run-time checks that are redundant. For instance, constant loop bounds allow a loop index to be treated as a constrained subtype, possibly obviating range and subscript checks involving the index.
  - Impact of language design on code quality. Positive impacts:
    - Named constants
    - Operator assigns (such as += in C). Allow redundant computations to be identified easily.
    - case statement, which generate significantly better code than equivalent if statement
    - Protected loop indices, which can be stored in registers and can often be guaranteed to be limited to a fixed range.
    - Restricted jumps and gotos, which make flow analysis easier.

Code optimization during various phases

- Language features that produce poor code or inhibit various optimizations:
  - By-name parameters
  - Function that have side effects, which may make code elimination or code movement impossible
  - Alias creation, which can make redundant expression analysis very difficult
  - Exceptions, which can cause unexpected (and invisible) jumps to handlers that may have side effects.
- IR representation optimizations:
  - Constant folding, Copy propagation, Reduction in strength, Inlining, Common sub-expressions
  - Loop-invariant, reduction in strength, loop unrolling,
  - Dead code elimination, Code motion.
- Code generation optimizations:
  - Careful allocation of registers
  - Thorough use of instruction sets
  - Thorough use of hardware addressing modes
  - Exploitation of special hardware considerations

Scope of optimization

- Scope of optimization can be local, global, and inter-procedural.
- **Local**: Usually applied to *straight-line segments of code*. (A basic block).
  - Constant folding
  - Copy propagation
  - Reduction in strength
  - Substitution of inline-code
- **Global**: Optimizations that extend beyond basic blocks.
  - More difficult. Usually requires a technique called **data flow analysis**, which attempts to collect information across jump boundaries.
- **Inter-procedural**: Optimizations that extend beyond boundaries of procedures of entire program. Much much more difficult:
  - analyze various parameter passing mechanisms;
  - possibility of non-local variable accesses;
  - may need to compute simultaneous information on all procedures that might call each other;
  - possibility of separate compilation

Principal sources of optimization

- Register allocation:
  - Good usage of registers important. Reduces the time it takes to go to memory to pick up information (whether on stack/heap etc.)
  - Problem: fixed number of registers vs. large number of variables. An optimization problem.
  - Two techniques used when designing microprocessors:
    - Define efficient memory operations. Do not need to depend on a very efficient register allocator.
    - Define large collection of registers (32, 64, 128) so that register allocation problem is easier. (Example: RISC chips).
- Unnecessary Operations: Avoid generating expressions that will not be needed.

  Approaches differ from simple local searches to searches across all programs.

Local optimization: Remove unnecessary ops

- **Common sub-expression elimination**: Remove many occurrences of an expression by its value (constraint: the value should not change across various occurrences).

```
Original code                Transformed code
a = (b + c)*m;                 T1 = b + c; a = T1*m;
x = b + c;                     x = T1;
y = (b + c) * z;               y = T1 * z;
```

- **Dead code elimination**: Dead code is code that is never executed or that does nothing useful.

```
      Original code                Transformed code
      T1 := k
       ...                           ....
      x := x + T1                  x := x + k
      y := x - T1                  y := x - k
 }                             }
```

- **Unnecessary jump elimination:**

```
Original code                Transformed code
x = 1;                       x = 1;
y = 2;                       y = 2;
if (x < y) jmp L1;           jmp L2;
jmp L2;                     L1: ...
L1: ...
}
```

Local optimization: Transform costly ops

- **Strength reduction**: Replace an expensive operation by a cheaper one.

```
Original code      Transformed code
x := x*2;            shift left (x);
x := y^3;            x := y * y * y;
```

- **Constant folding:** evaluate constant expressions at compile time. Can be complex in expressions when some of the components are constants. Example: lgth, amt: constant. Fold by reordering.

```
Original code                Transformed code
x := lgth * (b + c/a)* amt;  x := lgth*amt(b + c/a)
```

  Reordering may be problematic if numbers are floats.

  What if a variable is assigned once.. Almost like a constant. Called **Constant propagation** and needs to be done globally.

- **Procedure call**: Expensive to make procedure calls (save register states, build AR etc.). Two approaches to minimizing cost:
  - Procedure inlining: replace procedure call with the code of body.
  - Recognize tail recursion and replace it with gotos.

```
Original code                Transformed code
int gcd(int u, int v) {      int gcd(int u, int v) {
 if (v == 0) return u;        begin: if (v == 0) return u;
 else return gcd (v, u%v);     else { int t1 = v; int t2 = u%v ;
}                                    u = t1; v = t2;
                                     goto begin;
                              }
```

Code Optimization - cont'd

- Much optimization techniques depend on predicting program's behavior:
  - Collection information about variables, values, procedures
  - How are expressions used/reused?
  - Do variables remain constant or change value?

  Note: Compiler must make worst case assumptions about information it collects or risk generating incorrect code. Called **Conservative estimation** of program information.
- Another approach: Use statistical behavior about a program. Gather statistics through actual executions and use that to
  - predict which paths are most likely to be taken,
  - which procedures are most likely to be called often,
  - which sections of code are likely to be executed frequently.

  Use this information to adjust jump structure, loops, and procedure code to minimize execution speed for most commonly occurring executions.

Local optimization: Loop Optimization

- Programs spend 90% of time in 10% of code. (Mostly in loops) so it makes sense to optimize this portion of code...
- Factoring loop invariant expressions: Replace invariant expression from within the loop:

```
  Original code                Transformed code
for k := 1 to 1000 do          fact := 2*(p-q);
   c[k] := 2*(p - q)*(n-k+1)  denom := sqr(n) + n;
           / (sqr(n)+n);       for k := 1 to 1000 do
                                 c[k] := fact*(n-k+1)/denom;
```

  Compiler needs to move code so it needs to determine if some expression is dependent on loop indices.
- Reduction in strength: Replace more expensive operations by less expensive ones.

```
  Original code            Transformed code
for i := 1 to 1000 do        i := 1;  T1 := i - 1;
 sum := sum + a[i];          T2 := 4 * i; T3 := a[T2];
                             ...
                             i := i + 1;
    }
```

- Loop unrolling:

```
  Original code            Transformed code
for i := 1 to 20 do        for i := 1 to 20 do
  begin                      begin
  for j := 1 to 2 do           write(x[i,1], x[i,2]);
    write(x[i, j]);          end;
  end;
```

  Removes the overhead of setting up loop. Also, more optimization can be applied to the basic block.
- Loop fusion: combine two loops to create one loop.

Global Optimization

- Dead code elimination: Dead code is code that is never executed or that does nothing useful.
  May appear from copy propagation:

```
      T1 := k
         ...
      x := x + T1
      y := x - T1
         ...
```

  by

```
         ...
      x := x + k
      y := x - k
         ...
```

- Code motion: Used for optimizing code size.

```
   case p of
      1:  c := a + b * d;
      2:  m := b*d - r;
      3:  f := a - b*d;
   end;
```

  Replace by

```
T1 := b*d;
case p of
   1:  c := a + T1;
   2:  m := T1 - r;
end;
```