CODE OPTIMIZATION

- The code produced by straightforward compiling can often be made to run faster, take less space, or consume less energy by program transformations that are traditionally called 'optimizations'.
- Classifications
  - Machine independent, machine dependent
  - High-level, medium-level, machine-level
  - Scalar, parallel
- Optimization techniques
  - Using algebraic identities
  - Peephole optimization
  - Using dataflow information
- Criteria for code improving transformations
  - A transformation must preserve the meaning of programs
  - A transformation must, on the average, speed up programs
- Organization of a code optimizer
  - A code optimizer consists of
    - control-flow analysis
    - data-flow analysis
    - application of transformation
- Scalar transformations
  - Constant propagation
  - Constant folding
  - Dead code elimination
  - Unreachable code elimination
  - Copy propagation
  - Common subexpression elimination
  - Loop invariant code motion

- Control Flow Analysis
  - Basic blocks
  - Flow graphs
  - Identification of loops.
- Data-flow Analysis
  - Data-flow information can be collected by setting up and solving systems of equations that relate information at various points in a program. A typical equation has the form

    $$out[S] = gen[S] \cup (in[S] - kill[S])$$

    and can be read as

    "the information at the end of a statement is either generated within the statement, or enters at the beginning and is not killed as control flows through the statement."

    Such equations are called data-flow equations.
- The details of how data-flow equations are set up and solved depend on three factors:
  1. The notion of 'generating' and 'killing' depend on the desired information, i.e. on the data-flow analysis problem to be solved.
  2. Since data flows along control paths, data-flow analysis is affected by the control constructs in a program. In general, equations are set up at the level of basic blocks rather than statements, because blocks do have unique end points.
  3. There are subtleties that go along with such statements as procedure calls, assignments through pointer variables, and assignments to array variables.
- Iterative algorithm for 'Reaching Definitions':
  - Assuming that 'gen' and 'kill' have been computed for each block, we can create two groups of equations, shown below, that relate 'in' and 'out', information at input and output points of a basic block respectively.

- Algorithm : Reaching Definitions
  - Input : A flow graph for which 'kill' and 'gen' have been computed for each block B.
  - Output : IN [B] and OUT [B], the set of definition reaching the entry and exit of each block B of the flow graph.

1. OUT [ENTRY] = ϕ ;
2. for (each basic block B other than ENTRY) OUT [B] = ϕ ;
3. while (changes to any OUT occur)
4. for (each basic block B other than ENTRY)
5. { IN [B] = $\bigcup_{P \in \text{predecessor of } B}$ OUT [P] ;
6. OUT [B] = gen [B] $\cup$ (IN [B] - kill [B]); }

- The algorithm propagates definitions as far as they will go without being killed, thus simulating all possible executions of the program.
- Algorithm will eventually halt, because for every B, OUT [B] never shrinks ; once a definition is added, it stays there for ever. Since the set of all definitions is finite, eventually there must be a pass of the while loop during which nothing is added to any OUT, and the algorithm then terminates. We are safe terminating then because if the OUTs have not changed, the INs will not change on the next pass. And if the INs do not change, the OUTs cannot, so on all subsequent passes there can be no changes.
- The number of nodes in the flow graph is an upper bound on the number of times around the while-loop. The reason is that if a definition reaches a point, it can do so along a cycle-free path, and the number of nodes in a flow graph is an upper bound on the number of nodes in a cycle-free path. Each time around the while-loop, each definition progresses by at least one node along the path, depending on the order of visit of nodes.

- Algorithm : Available Expressions
  - Input : A flow graph with $kill_B$ and gen_B computed for each block B.
  - Output : IN[B] and OUT[B], the set of expressions available at the entry and exit of each block B of the flow graph.

```
OUT [ENTRY] = ϕ
for (each basic block B other than ENTRY)
     OUT [B] = U ;
while ( changes to any OUT occur)
     for (each basic block B other than ENTRY)
     {
          IN [B] = ∩_{P ∈ predecessor of B} OUT [P] ;

          OUT[B] = gen [B] ∪ (IN [B] - kill [B]);
     }
```

- Here, U is the set of all expressions in the program.
- Example:

Statement	Available Expressions
a = b + c	
	{b + c}
b = a - d	
	{a - d}
e = b + c	
	{a - d}
d = a - d	
	ϕ