Improving Code Generation
Better code generation requires greater context.

- Inter-procedural flow analysis
- Over the program:
  - Global register allocation, register coloring
- Over procedures:
  - Register tracking with last-use information
  - Common subexpression elimination
- Over basic blocks:
  - Optimal ordering of subtrees
- Over expressions:
Better code generation requires information about
points of definition and points of use of variables

Basic Blocks
A basic block is formed out of a leader and all
instructions that follow, up to but not including the
next leader.

- Any instruction that is the target of a jump is a leader
- Any instruction that is the target of a jump is a leader
- Any instruction following a call to a procedure is a leader
- Any instruction following a call to a basic block (quadruple) is a leader
- Call the first instruction in the program is a leader
- The first instruction in the program is a leader

Decomposition into Basic Blocks:

To partition a program into basic blocks:
Transformations on Basic Blocks

- Common subexpression elimination: recognize redundant re-computations. Replace with a single temporary.
- Interchanging statements, for better scheduling
- Renaming of temporaries, for better register usage
- Dead-code elimination: recognize computations whose results are never used. Remove associated quadruples

All of the above require symbolic execution of the basic block to obtain definition/use information.

Transformations on Basic Blocks
Within block, computed by a single backwards pass over quadruples and symbol table.

Next-use information is an annotation over quadruples.

Next-use information is an annotation over quadruples.

Next-use information is an annotation over quadruples.

If \( x \) is re-computed at \( k \), the value computed at \( k \) has no further use, and can be discarded (i.e., register re-used).

If \( x \) is computed in quadruple \( j \), and is an operand of quadruple \( i \), its value must be preserved (register or memory) until \( j \).

If \( x \) is computed in quadruple \( j \), and is an operand of quadruple \( i \), its value must be preserved (register or memory) until \( j \).
next-use of \( y, z \) is 9, mark both as live.

Mark \( x \) as dead in symbol table (previous value has no next-use)

Record next uses of \( x, y, z \) from symbol table into quadruple

For quadruple \( q: x = y \ op \ z \)

On exit from block, all temporaries are dead (no next-use)

Operand next-use (later quadruple number)

Operand liveness (boolean)

Informaition:

Each operand in a quadruple and symbol table carries additional

Use symbol table to annotate status of variables

Computing next-use
Dead-Code Elimination (within basic blocks)

Remove quadruple "x := y op z" if x is dead.

Examples:

- If x is a temporary, not referenced in any later quadruple.
- Dead. can be removed.
- No reference to x

After elimination, needs to update/recompute next-use information.
Goal is to minimize use of registers and memory references.

Doubly linked data structure:

Procedure getReg determines "optimal" choice to hold result of next quadruple.

For each variable, indicate location of current value:
- register descriptor: set of variables with equal values
- memory and/or registers: address descriptor

For each register, indicate current contents:
- register descriptor

Register Allocation over Basic Block:

Tracking
Choose variable whose next use is farthest away. •
memory (spill), and use register.
else find a register that holds a live variable, store variable in
memory, use $R_i$.

if $y$ is in $R_i$, $R_i$ contains no other variable, and $y$ is also in
else, if there is a register $R_i$ that holds a dead variable, use it.
else, if there is an available register $R_j$, use it.
else, if there is an available register operation.
else, try the same for $z$, provided architecture supports

If $y$ is in $R_i$, $R_i$ contains no other variable, $y$ is not live, and there
for quadruple $x := y$ op $z$,

Getreag: Heuristics

A. Pnueli
Using getreg

For \( x := y \) or \( z \):

1. Call getreg to obtain target register \( R \).
2. Find current location of \( y \), generate load into register \( R \) if in memory, update address descriptor for \( y \).
3. Ditto for \( z \), except that should use register other than \( R \).
4. Emit instruction.
5. Update register descriptor for \( R \) to indicate it holds \( x \).
6. Update address descriptor for \( x \) to indicate it resides in \( R \).

For \( x := y \):

1. Single load, register descriptor indicates that both \( x \) and \( y \) are in \( R \).

On block exit, store registers that contain live values which have no copy in memory.
Using getreg

\[
\begin{align*}
\text{Statement} & : n + \alpha =: p \\
\text{Register} & : n + \gamma =: \alpha \\
\text{Register} & : \gamma - \alpha =: n \\
\text{Register} & : \gamma - \alpha =: \gamma \\
\end{align*}
\]

The assignment \((\gamma - \alpha) + (\gamma - \alpha) + (\gamma - \alpha) =: p\) can be translated to getreg.
Computing Dependencies in a Basic Block: The DAG

A Pnueli

Honors Compilers, NYU, Fall '09

Improved Code Generation

Computing Dependencies in a Basic Block:

A. Pnueli

Honors Compilers, NYU, Fall '09

Intermediate code optimization:

Leaf nodes are labeled with identifiers and constants.

Internal nodes are labeled with operators and identifiers.

Leaves are labeled with identifiers and constants.

Intermedi ate code optimization:

Use directed acyclic graph (dag) to recognize common subexpressions and remove redundant quadruples.

basic block ↔ improved dag ↔ basic block ↔ improved dag

Use directed acyclic graph (dag) to recognize common subexpressions and remove redundant quadruples.

Intermedi ate code optimization:

Leaf nodes are labeled with identifiers and constants.

Internal nodes are labeled with operators and identifiers.

Leaves are labeled with identifiers and constants.

Intermediate code optimization:

Use directed acyclic graph (dag) to recognize common subexpressions and remove redundant quadruples.

basic block ↔ improved dag ↔ basic block ↔ improved dag

Use directed acyclic graph (dag) to recognize common subexpressions and remove redundant quadruples.
A. Pnueli

Forward pass over basic block

\[ \text{Add } x \text{ to list of labels for node which currently holds } y \]

\[ \text{Add } x \text{ to list of labels for new node } \]

\[ \text{Create new node for } \text{op}, \text{ or find an existing one with descendants } y, z \text{ (need hash scheme)} \]

\[ \text{Find node labeled } y, \text{ or create one } \]

\[ \text{Find node labeled } z, \text{ or create one } \]

\[ \text{For } x : = y \]

\[ \text{For } x : = y \text{ op } z \]

DAG Construction
Example: dot product

\[
\text{prod} := 0; \\
\text{for } j \in 1 \ldots 20 \text{ loop} \\
\quad \text{prod} := \text{prod} + a(j) \times b(j); \\
\text{end loop;}
\]

- Assume 4-byte integer
- Quadruples:
  - end loop;
  - prod := prod + \text{a}(j) \times \text{b}(j); \\
  - prod := 0;

Quadruples:
- redundant
- basic block leader
- basic block leader
- basic block leader
- start: 
  - T1 := 4
  - T2 := \text{a}(T1)'
  - T3 := 4 \times \text{!}'
  - T4 := \text{b}(T1)'
  - T5 := T2 \times T4'
  - T6 := T1
  - prod := T6;
  - T7 := j + 1
  - j := T7;
  - if j \geq 20 goto start

Honors Compilers, NYU, Fall, 2009
Common subexpressions identified

DAG for Body of Loop
Fewer quadruples, fewer temporaries

\[
\text{if } j \geq 20 \text{ goto start}
\]

! + 1;
prod + \_5;
prod = \_5 * \_4;
T2 = b[\_1];
T4 = a[\_1];
T2 = 4 * \_1;
start: \_1 = \_1

Prefer the label of a live variable over a temporary

A node without a label is a dead value

Any topological sort of the DAG is a legal evaluation order

From DAG to Improved Block
Programmers don’t produce common subexpressions, code generators do!

\[ A, B \text{ array}\{\text{lo1..hi1, lo2..hi2}\}; \]

Components size \( w \) bytes

- \( w \) is often a power of 2 (peephole optimization).
- \( w \) is an invariant (loop optimization).
- Can reduce to 1 with a DAG.

\[
\text{base} = a + (\text{lo1} - 1) \times (\text{hi2} - \text{lo2} + 1) + (\text{lo2} - 1) \times w
\]

The following requires 19 quadruples:

\[
\text{for } k \text{ in } \text{lo2..hi2} \text{ loop}
\]

\[
A[i,k] := B[i,k] + 1;
\]

\[
\text{end loop;}
\]

\[
\text{A[i,k]} = B[i,k] + 1;
\]

\[
\text{for } k \text{ in } \text{lo2..hi2} \text{ loop}
\]

\[
\text{The following requires } 19 \text{ quadruples:}
\]

\[
\text{base} = a + (\text{lo1} - 1) \times (\text{hi2} - \text{lo2} + 1) + (\text{lo2} - 1) \times w
\]

\[
\text{A[i,k]} \text{ is an location:}
\]

\[
A[i,k] \text{ is at location:}
\]

\[
A, B: \text{array}\{\text{lo1..hi1, lo2..hi2}\};
\]
Beyond Basic Blocks: Data Flow Analysis

- Requires complex data structures and algorithms
  - Loop invariant computations
  - Live-dead analysis
  - Common subexpressions elimination
  - Constant folding

Algorithms on graphs:
- Can compute global properties of programs as iterative
- Basic blocks are nodes in the flow graph
Register assignment is equivalent to graph coloring

Corresponding variables overlap

There is an edge between two nodes if the lifetime of the

Each variable is a node in the graph

Lifetime information is translated into interference graph:

Lifetime overlap.

Two variables cannot be assigned the same register if their

Instructions in program.

To reuse registers, need to know lifetime of variables (set of

Registers throughout program.

Optimal use of registers in subprograms: Keep all variables in

Coloring

Using Global Information: Register
Given a graph and a set of $N$ colors, assign a color to each vertex so that not two connected vertices are colored by the same color.

Problem is NP-Complete

Fast heuristics algorithm (Chaitin) is usually linear:

- If at any point a node has more than $N - 1$ neighbors, need to free a register (spill). Can then remove node and continue.
- Iterate until graph is empty, then assign colors in inverse order of neighbors.
- Any node with fewer than $N - 1$ neighbors is colorable, so can be deleted from graph. Start with node with smallest number of neighbors.
- If at any point a node has more than $N - 1$ neighbors, need to free a register (spill). Can then remove node and continue.

Honors Compilers, NYU, Fall, 2009
Example

Order of removal: B, C, A', E, F, D

Assume 3 colors are available: assign colors in reverse order, constrained by already colored nodes.

- D (no constraint)
- F (D)
- E (D)
- A (F, E)
- C (D, A)
- B (A, C)
Better Approach to Spilling

- Use loop structure to estimate usage.
- Spill variables with lowest usage count.
- Need to place $N - R$ variables in memory.
- Compute required number of colors in second pass: $R$. 

A. Pnueli