Code Generation
Part II

Chapter 8
(1st ed. Ch.9)
Flow Graphs

- A *flow graph* is a graphical depiction of a sequence of instructions with control flow edges

- A flow graph can be defined at the intermediate code level or target code level

```
MOV 1,R0
MOV n,R1
JMP L2
L1: MUL 2,R0
   SUB 1,R1
L2: JMPNZ R1,L1

MOV 0,R0
MOV n,R1
JMP L2
L1: MUL 2,R0
   SUB 1,R1
L2: JMPNZ R1,L1
```
Basic Blocks

- A basic block is a sequence of instructions
  - Control enters through the first instruction
  - Control leaves the block without branching, except possibly at the last instruction

```
MOV 1, R0
MOV n, R1
JMP L2

L1: MUL 2, R0
    SUB 1, R1
L2: JMPNZ R1, L1
```

```
MOV 1, R0
MOV n, R1
JMP L2

L1: MUL 2, R0
    SUB 1, R1
L2: JMPNZ R1, L1
```
Basic Blocks and Control Flow Graphs

• A control flow graph (CFG) is a directed graph with basic blocks $B_i$ as vertices and with edges $B_i \rightarrow B_j$ iff $B_j$ can be executed immediately after $B_i$. 

```plaintext
MOV 1,R0
MOV n,R1
JMP L2
L1: MUL 2,R0
SUB 1,R1
L2: JMPNZ R1,L1
```
Successor and Predecessor Blocks

- Suppose the CFG has an edge $B_1 \rightarrow B_2$
  - Basic block $B_1$ is a predecessor of $B_2$
  - Basic block $B_2$ is a successor of $B_1$

```
MOV 1, R0
MOV n, R1
JMP L2

L1: MUL 2, R0
SUB 1, R1

L2: JMPNZ R1, L1
```
Partition Algorithm for Basic Blocks

*Input*: A sequence of three-address statements
*Output*: A list of basic blocks with each three-address statement in exactly one block

1. Determine the set of *leaders*, the first statements if basic blocks
   a) The first statement is the leader
   b) Any statement that is the target of a goto is a leader
   c) Any statement that immediately follows a goto is a leader
2. For each leader, its basic block consist of the leader and all statements up to but not including the next leader or the end of the program
Loops

• A loop is a collection of basic blocks, such that
  – All blocks in the collection are strongly connected
  – The collection has a unique entry, and the only way to reach a block in the loop is through the entry
Loops (Example)

Strongly connected components:

SCC={ \{B2,B3\}, \{B4\} }

Entries:
B3, B4
Equivalence of Basic Blocks

- Two basic blocks are (semantically) equivalent if they compute the same set of expressions.

\[
\begin{align*}
\text{b} & := 0 \\
\text{t1} & := \text{a} + \text{b} \\
\text{t2} & := \text{c} \times \text{t1} \\
\text{a} & := \text{t2}
\end{align*}
\]

\[
\begin{align*}
\text{a} & := \text{c} \times \text{a} \\
\text{b} & := 0
\end{align*}
\]

Blocks are equivalent, assuming \text{t1} and \text{t2} are dead: no longer used (no longer live).
Transformations on Basic Blocks

• A *code-improving transformation* is a code optimization to improve speed or reduce code size

• *Global transformations* are performed across basic blocks

• *Local transformations* are only performed on single basic blocks

• Transformations must be safe and preserve the meaning of the code
  – A local transformation is safe if the transformed basic block is guaranteed to be equivalent to its original form
Common-Subexpression Elimination

- Remove redundant computations

\[
\begin{align*}
a & := b + c \\
b & := a - d \\
c & := b + c \\
d & := a - d \\
\end{align*}
\]

\[
\begin{align*}
t1 & := b * c \\
t2 & := a - t1 \\
t3 & := b * c \\
t4 & := t2 + t3 \\
\end{align*}
\]

\[
\begin{align*}
a & := b + c \\
b & := a - d \\
c & := b + c \\
d & := b \\
\end{align*}
\]

\[
\begin{align*}
t1 & := b * c \\
t2 & := a - t1 \\
t4 & := t2 + t1 \\
\end{align*}
\]
Dead Code Elimination

- Remove unused statements

\[
\begin{align*}
b &:= a + 1 \\
a &:= b + c \\
... \\
\end{align*}
\]

Assuming \( a \) is \textit{dead} (not used)

\[
\begin{align*}
\text{if true goto L2} \\
b &:= x + y \\
... \\
\end{align*}
\]

Remove unreachable code
Renaming Temporary Variables

• Temporary variables that are dead at the end of a block can be safely renamed

\[
\begin{align*}
    t1 & := b + c \\
    t2 & := a - t1 \\
    t1 & := t1 * d \\
    d & := t2 + t1
\end{align*}
\]

\[
\begin{align*}
    t1 & := b + c \\
    t2 & := a - t1 \\
    t3 & := t1 * d \\
    d & := t2 + t3
\end{align*}
\]

Normal-form block
Interchange of Statements

• Independent statements can be reordered

\[
\begin{align*}
t1 & := b + c \\
t2 & := a - t1 \\
t3 & := t1 \times d \\
d & := t2 + t3
\end{align*}
\]

\[
\begin{align*}
t1 & := b + c \\
t3 & := t1 \times d \\
t2 & := a - t1 \\
d & := t2 + t3
\end{align*}
\]

Note that normal-form blocks permit all statement interchanges that are possible
Algebraic Transformations

• Change arithmetic operations to transform blocks to algebraic equivalent forms

\[
\begin{align*}
\text{t1} & := a - a \\
\text{t2} & := b + \text{t1} \\
\text{t3} & := 2 \times \text{t2}
\end{align*}
\]

\[
\begin{align*}
\text{t1} & := 0 \\
\text{t2} & := b \\
\text{t3} & := \text{t2} \ll 1
\end{align*}
\]
Next-Use

- Next-use information is needed for dead-code elimination and register assignment
- Next-use is computed by a backward scan of a basic block and performing the following actions on statement
  \[ i: \quad x := y \text{ op } z \]
  - Add liveness/next-use info on \( x, y, \) and \( z \) to statement \( i \)
  - Before going up to the previous statement (scan up):
    - Set \( x \) info to “not live” and “no next use”
    - Set \( y \) and \( z \) info to “live” and the next uses of \( y \) and \( z \) to \( i \)
Next-Use (Step 1)

\[ i: \quad b := b + 1 \]

\[ j: \quad a := b + c \]

\[ k: \quad t := a + b \quad [ \text{live}(a) = \text{true}, \text{live}(b) = \text{true}, \text{live}(t) = \text{true}, \text{nextuse}(a) = \text{none}, \text{nextuse}(b) = \text{none}, \text{nextuse}(t) = \text{none} ] \]

Attach current live/next-use information
Because info is empty, assume variables are live
(Data flow analysis Ch.10 can provide accurate information)
Next-Use (Step 2)

\[ i: \quad b := b + 1 \]

\[ j: \quad a := b + c \begin{array}{ll}
\text{live}(a) &= \text{true} \\
\text{nextuse}(a) &= k \\
\text{live}(b) &= \text{true} \\
\text{nextuse}(b) &= k \\
\text{live}(t) &= \text{false} \\
\text{nextuse}(t) &= \text{none}
\end{array} \]

\[ k: \quad t := a + b \begin{array}{ll}
\text{live}(a) &= \text{true}, \text{live}(b) = \text{true}, \text{live}(t) = \text{true}, \\
\text{nextuse}(a) &= \text{none}, \text{nextuse}(b) = \text{none}, \text{nextuse}(t) = \text{none}
\end{array} \]

Compute live/next-use information at \( k \)
Next-Use (Step 3)

\[ \text{i: } b := b + 1 \]

\[ \text{j: } a := b + c \quad \text{[live}(a) = \text{true}, \text{live}(b) = \text{true}, \text{live}(c) = \text{false}, \text{nextuse}(a) = k, \text{nextuse}(b) = k, \text{nextuse}(c) = \text{none}] } \]

\[ \text{k: } t := a + b \quad \text{[live}(a) = \text{true}, \text{live}(b) = \text{true}, \text{live}(t) = \text{true}, \text{nextuse}(a) = \text{none}, \text{nextuse}(b) = \text{none}, \text{nextuse}(t) = \text{none}] \]

Attach current live/next-use information to \(j\)
Next-Use (Step 4)

$i$: \quad b := b + 1

\begin{align*}
  \text{live}(a) &= \text{false} \quad \text{nextuse}(a) = \text{none} \\
  \text{live}(b) &= \text{true} \quad \text{nextuse}(b) = j \\
  \text{live}(c) &= \text{true} \quad \text{nextuse}(c) = j \\
  \text{live}(t) &= \text{false} \quad \text{nextuse}(t) = \text{none}
\end{align*}

$j$: \quad a := b + c

\begin{align*}
  \text{live}(a) &= \text{true}, \quad \text{live}(b) = \text{true}, \quad \text{live}(c) = \text{false}, \\
  \text{nextuse}(a) &= k, \quad \text{nextuse}(b) = k, \quad \text{nextuse}(c) = \text{none}
\end{align*}

$k$: \quad t := a + b

\begin{align*}
  \text{live}(a) &= \text{true}, \quad \text{live}(b) = \text{true}, \quad \text{live}(t) = \text{true}, \\
  \text{nextuse}(a) &= \text{none}, \quad \text{nextuse}(b) = \text{none}, \quad \text{nextuse}(t) = \text{none}
\end{align*}

Compute live/next-use information \( j \)
Next-Use (Step 5)

\[ i: \quad b := b + 1 \quad [ \text{live}(a) = \text{false}, \text{live}(b) = \text{true}, \text{live}(c) = \text{true}, \text{nextuse}(a) = \text{none}, \text{nextuse}(b) = j, \text{nextuse}(c) = j ] \]

\[ j: \quad a := b + c \quad [ \text{live}(a) = \text{true}, \text{live}(b) = \text{true}, \text{live}(c) = \text{false}, \text{nextuse}(a) = k, \text{nextuse}(b) = k, \text{nextuse}(c) = \text{none} ] \]

\[ k: \quad t := a + b \quad [ \text{live}(a) = \text{true}, \text{live}(b) = \text{true}, \text{live}(t) = \text{true}, \text{nextuse}(a) = \text{none}, \text{nextuse}(b) = \text{none}, \text{nextuse}(t) = \text{none} ] \]

Attach current live/next-use information to \( i \)
A Code Generator

- Generates target code for a sequence of three-address statements using *next-use* information
- Uses `getreg` to assign registers to variables
- For instruction \( x := y \text{ op } z \)
  
  \( \text{getreg}(y, z) \) returns a location (register) for \( x \)
- Results are kept in *registers* as long as possible:
  - Result is needed in another computation
  - Register is kept up to a procedure call or end of block
- Check if operands of three-address code are available in registers
The Code Generation Algorithm

- For each statement $x := y \text{ op } z$
  1. Set location $L = \text{getreg}(y, z)$ to get register for $x$
  2. If $y \notin L$ then generate
     \[ \text{MOV } y', L \]
     where $y'$ denotes one of the locations where the value of $y$ is available (choose register if possible)
  3. Generate
     \[ \text{OP } z', L \]
     where $z'$ is one of the locations of $z$;
     Update register/address descriptor of $x$ to include $L$
  4. If $y$ and/or $z$ has no next use and is stored in register, update register descriptors to remove $y$ and/or $z$
Register and Address Descriptors

- A *register descriptor* keeps track of what is currently stored in a register at a particular point in the code, e.g. a local variable, argument, global variable, etc.
  
  \[ \text{MOV } a, R0 \quad \text{"R0 contains } a\text{"} \]

- An *address descriptor* keeps track of the location where the current value of the name can be found at runtime, e.g. a register, stack location, memory address, etc.

  \[ \text{MOV } a, R0 \quad \text{MOV } R0, R1 \quad \text{"a in } R0\text{ and } R1\text{"} \]
The \textit{getreg} Algorithm

- To compute \textit{getreg}(y, z)
  
  1. If \(y\) is stored in a register \(R\) and \(R\) only holds the value \(y\), and \(y\) has no next use, then return \(R\);
     Update address descriptor: value \(y\) no longer in \(R\)
  
  2. Else, return a new empty register if available
  
  3. Else, find an occupied register \(R\);
     Store contents (\textit{register spill}) by generating
     \textbf{MOV} \(R, M\)
     for every \(M\) in address descriptor of \(y\);
     Return register \(R\)
  
  4. Return a memory location
## Code Generation Example

<table>
<thead>
<tr>
<th>Statements</th>
<th>Code Generated</th>
<th>Register Descriptor</th>
<th>Address Descriptor</th>
</tr>
</thead>
<tbody>
<tr>
<td>t := a - b</td>
<td>MOV a,R0</td>
<td>Registers empty</td>
<td>t in R0</td>
</tr>
<tr>
<td></td>
<td>SUB b,R0</td>
<td>R0 contains t</td>
<td></td>
</tr>
<tr>
<td>u := a - c</td>
<td>MOV a,R1</td>
<td>R1 contains u</td>
<td>u in R1</td>
</tr>
<tr>
<td></td>
<td>SUB c,R1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>v := t + u</td>
<td>ADD R1,R0</td>
<td>R0 contains v</td>
<td>u in R1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R1 contains u</td>
<td>v in R0</td>
</tr>
<tr>
<td>d := v + u</td>
<td>ADD R1,R0</td>
<td>R0 contains d</td>
<td>d in R0</td>
</tr>
<tr>
<td></td>
<td>MOV R0,d</td>
<td></td>
<td>d in R0 and memory</td>
</tr>
<tr>
<td>live(d)=true</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>all other dead</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Register Allocation and Assignment

- The `getreg` algorithm is simple but sub-optimal
  - All live variables in registers are stored (flushed) at the end of a block

- *Global register allocation* assigns variables to limited number of available registers and attempts to keep these registers consistent across basic block boundaries
  - Keeping variables in registers in looping code can result in big savings
Allocating Registers in Loops

- Suppose loading a variable $x$ has a cost of 2.
- Suppose storing a variable $x$ has a cost of 2.
- Benefit of allocating a register to a variable $x$ within a loop $L$ is

$$\sum_{B \in L} (use(x, B) + 2 \cdot live(x, B))$$

where $use(x, B)$ is the number of times $x$ is used in $B$ and $live(x, B) = true$ if $x$ is live on exit from $B$. 
Global Register Allocation with Graph Coloring

- When a register is needed but all available registers are in use, the content of one of the used registers must be stored (spilled) to free a register
- Graph coloring allocates registers and attempts to minimize the cost of spills
- Build a conflict graph (interference graph)
- Find a $k$-coloring for the graph, with $k$ the number of registers
Register Allocation with Graph Coloring: Example

```plaintext
a := read();
b := read();
c := read();
a := a + b + c;
if (a < 10) {
    d := c + 8;
    write(c);
} else if (a < 20) {
    e := 10;
    d := e + a;
    write(e);
} else {
    f := 12;
    d := f + a;
    write(f);
}
write(d);
```
Register Allocation with Graph Coloring: Live Ranges

```
a := read();
b := read();
c := read();
a := a+b+c;
a < 20

f := 12;
d := f+a;
write(f);
e := 10;
d := e+a;
write(e);
d := c+8;
write(c);
```

Interference graph:
connected vars have overlapping ranges
Register Allocation with Graph Coloring: Solution

Interference graph

Solve

Three registers:

\[ a = r_2 \]
\[ b = r_3 \]
\[ c = r_1 \]
\[ d = r_2 \]
\[ e = r_1 \]
\[ f = r_1 \]

```
r2 := read();
r3 := read();
r1 := read();
r2 := r2 + r3 + r1;
if (r2 < 10) {
    r2 := r1 + 8;
    write(r1);
} else if (r2 < 20) {
    r1 := 10;
    r2 := r1 + r2;
    write(r1);
} else {
    r1 := 12;
    r2 := r1 + r2;
    write(r1);
}
write(r2);
```
Peephole Optimization

• Examines a short sequence of target instructions in a window (*peephole*) and replaces the instructions by a faster and/or shorter sequence when possible
• Applied to intermediate code or target code
• Typical optimizations:
  – Redundant instruction elimination
  – Flow-of-control optimizations
  – Algebraic simplifications
  – Use of machine idioms
Peephole Opt: Eliminating Redundant Loads and Stores

• Consider
  \[
  \text{MOV R0,a}
  \]
  \[
  \text{MOV a,R0}
  \]

• The second instruction can be deleted, but only if it is not labeled with a target label
  – Peephole represents sequence of instructions with at most one entry point

• The first instruction can also be deleted if \(\text{live(a)}=\text{false}\)
Peephole Optimization: Deleting Unreachable Code

- Unlabeled blocks can be removed

```plaintext
if 0==0 goto L2
b := x + y
...
goto L2
b := x + y
...
```
Peephole Optimization: Branch Chaining

- Shorten chain of branches by modifying target labels

if a==0 goto L2

b := x + y
...
L2: goto L3

if a==0 goto L3

b := x + y
...
L2: goto L3
Peephole Optimization: Other Flow-of-Control Optimizations

• Remove redundant jumps

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

...
Other Peephole Optimizations

• *Reduction in strength*: replace expensive arithmetic operations with cheaper ones

  \[
  \begin{align*}
  &\cdots \\
  &a := x ^ 2 \\
  &b := y / 8 \\
  \end{align*}
  \quad \rightarrow 
  \begin{align*}
  &\cdots \\
  &a := x \times x \\
  &b := y \gg 3 \\
  \end{align*}
  \]

• Utilize machine idioms

  \[
  \begin{align*}
  &\cdots \\
  &a := a + 1 \\
  \end{align*}
  \quad \rightarrow 
  \begin{align*}
  &\cdots \\
  &\text{inc } a \\
  \end{align*}
  \]

• Algebraic simplifications

  \[
  \begin{align*}
  &\cdots \\
  &a := a + 0 \\
  &b := b \times 1 \\
  \end{align*}
  \quad \rightarrow 
  \begin{align*}
  &\cdots \\
  \end{align*}
  \]