CSE443
Compilers

Dr. Carl Alphonce
alphonce@buffalo.edu
343 Davis Hall
Announcements

- Recitations
  - F2F w/490 at 2:00 in Baldy 110
Phases of a compiler

Figure 1.6, page 5 of text

Target machine code generation
Code Transformations on basic blocks

- Local optimizations can be performed on code inside basic blocks.
- Represent code inside a basic block as a DAG.
- The basic blocks will themselves be connected to form a flow graph.
Example from last class

Figure 8.7 [p. 527]

1) \( i = 1 \)
2) \( j = 1 \)
3) \( t1 = 10 \times i \)
4) \( t2 = t1 + j \)
5) \( t3 = 8 \times t2 \)
6) \( t4 = t3 - 88 \)
7) \( a[t4] = 0.0 \)
8) \( j = j + 1 \)
9) if \( j \leq 10 \) goto (3)
10) \( i = i + 1 \)
11) if \( i \leq 10 \) goto (2)
12) \( i = 1 \)
13) \( t5 = i - 1 \)
14) \( t6 = 88 \times t5 \)
15) \( a[t6] = 1.0 \)
16) \( i = i + 1 \)
17) if \( i \leq 10 \) goto (13)
Identifying leaders

1) \( i = 1 \)
2) \( j = 1 \)
3) \( t_1 = 10 \times i \)
4) \( t_2 = t_1 + j \)
5) \( t_3 = 8 \times t_2 \)
6) \( t_4 = t_3 - 88 \)
7) \( a[t_4] = 0.0 \)
8) \( j = j + 1 \)
9) if \( j \leq 10 \) goto (3)
10) \( i = i + 1 \)
11) if \( i \leq 10 \) goto (2)
12) \( i = 1 \)
13) \( t_5 = i - 1 \)
14) \( t_6 = 88 \times t_5 \)
15) \( a[t_6] = 1.0 \)
16) \( i = i + 1 \)
17) if \( i \leq 10 \) goto (13)

Leaders are:
1. first instruction
2. the target of any jump
3. the instruction immediately after any jump
```plaintext
Flow Graph
Figure 8.9 [p. 530]

Entry and exit nodes added.
Jump targets replaced by block names.

B1: i = 1

B2: j = 1

B3:
- t1 = 10 * i
- t2 = t1 + j
- t3 = 8 * t2
- t4 = t3 - 88
- a[t4] = 0.0
- j = j + 1
- if j <= 10 goto B3

B4:
- i = i + 1
- if i <= 10 goto B2

B5: i = 1

B6:
- t5 = i - 1
- t6 = 88 * t5
- a[t6] = 1.0
- i = i + 1
- if i <= 10 goto B6
```

ENTRY

EXIT
1. For each variable in the block, create a node representing the variable’s initial value.

2. For each statement \( s \) in the block, create a node \( N \).

"The children of \( N \) are those nodes corresponding to statements that are the last definitions, prior to \( s \), of the operands used by \( s \)."
Constructing DAG for basic blocks

3. For each node representing a statement, label it with the operator applied.

4. For each node representing a statement, attach a list of the variables for which it is the last definition within the block.
Constructing DAG for basic blocks

5. For each node representing a statement, its children are the nodes that are the last definitions of the operands used in the statement.

6. Identify as output nodes those whose variables are live on exit from the block ("their values may be used later, in another block of the flow graph")
Example 8.10 [p. 534]

1) \( a = b + c \)
2) \( b = a - d \)
3) \( c = b + c \)
4) \( d = a - d \)
Example 8.10 [p. 534]

Apply the "value-number" method from section 6.1.1

1) \( a = b + c \)
2) \( b = a - d \)
3) \( c = b + c \)
4) \( d = a - d \)
Example 8.10 [p. 534]

Apply the "value-number" method from section 6.1.1

1) $a = b + c$
2) $b = a - d$
3) $c = b + c$
4) $d = a - d$
Example 8.10 [p. 534]

Apply the "value-number" method from section 6.1.1

1) \( a = b + c \)
2) \( b = a - d \)
3) \( c = b + c \)
4) \( d = a - d \)
Example 8.10 [p. 534]

1) \(a = b + c\)
2) \(b = a - d\)
3) \(c = b + c\)
4) \(d = a - d\)

Apply the "value-number" method from section 6.1.1.
Example 8.10 [p. 534]

Apply the "value-number" method from section 6.1.1

1) $a = b + c$
2) $b = a - d$
3) $c = b + c$
4) $d = a - d$
Example 8.10 [p. 534]

If $b$ is live on exit:

1) $a = b + c$
2) $b = a - d$
3) $c = b + c$
4) $d = b$

Diagram: |
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>---</td>
</tr>
<tr>
<td>---</td>
</tr>
<tr>
<td>---</td>
</tr>
<tr>
<td>---</td>
</tr>
</tbody>
</table>

- $b$, $d$
- $b_0$
- $c_0$
- $+$
- $-$
- $+$
- $-$
Example 8.10 [p. 534]

If \( b \) is not live on exit:

1) \( a = b + c \)
2) \( d = a - d \)
3) \( c = d + c \)
8.6 A Simple Code Generator [p. 542]

- algorithm focuses on generation of code for a single basic block
- generates code for each three address code instruction
- manages register allocations/assignment to avoid redundant loads/stores
Principal uses of registers

- operator operands must be in registers
- temporaries needed within block
- variables that span multiple blocks
- stack pointer
- function arguments
“We [...] assume that for each operator, there is exactly one machine instruction that takes the necessary operands in registers and performs that operation, leaving the result in a register. The machine instructions are of the form:

- LD reg, mem
- ST mem, reg
- OP reg, reg, reg” [p. 543]

OP rd, rs1, rs2 – where rs2 can be immediate (a constant) – c.f. Kris’s presentation on Monday
8.6.1 Register and Address Descriptors

A three-address instruction of the form:

\[ v = a \text{ op } b \]

we generate:

- `LD Rx, a`
- `LD Ry, b`
- `OP Rx, Rx, Ry`
- `ST Rx, v`
This results in many redundant loads and stores.

This may not make effective use of available registers.

Use two data structures:
- register descriptor
- address descriptor
register descriptor

"For each available register, a register descriptor keeps track of the variables names whose current value is in that register." [p. 543]
address descriptor

"For each program variable, an address descriptor keeps track of the location or locations where the current value of that variable can be found." [p. 543]
getReg function

"...getReg(I)...selects registers for each memory location associated with the three-address instruction I." [p. 544]
Example
(paraphrased from 8.6.2, page 544)

A three-address instruction of the form:
\[ v = a \text{ op } b \]

1. Use `getReg(v = a \text{ op } b)` to select registers for \( v \), \( a \) and \( b \): \( R_v \), \( R_a \), and \( R_b \) respectively.

2. If \( a \) is not already in \( R_a \), generate LD \( R_a, a' \) (where \( a' \) is one of the possibly many current locations of \( a \)).

3. Similarly for \( b \).

4. Generate OP \( R_v, R_a, R_b \)
"We assume getReg will always choose the same register for both x and y. If y is not already in that register Ry, then generate the machine instruction LD Ry, y. If y was already in Ry, we do nothing. It is only necessary that we adjust the register descriptor for Ry so that it includes x as one of the values found there." [p. 544]
Writing back to memory at end of block

At the end of a basic block we must ensure that live variables are stored back into memory.

"...for each variable x whose address descriptor does not say that is value is located in the memory location for x, we must generate the instruction ST x, R, where R is a register in which x's value exists at the end of the block." [p. 545]
Updating register descriptors (RD) and address descriptors (AD)

1. LD R, x
   (a) Set RD of R to only x
   (b) Add R to AD of x
2. ST x, R
   (a) Add &x to AD of x
3. OP Rx, Ry, Rz for x = y op z
   (a) Set RD of Rx to only x
   (b) Set AD of x to only Rx (&x not in AD of x!)
   (c) Remove Rx from the AD of any variable other than x
4. "When we process a copy statement x = y, after generating the load for y into register Ry, if needed, and after managing descriptors as for all load statement (per rule 1):" [p. 545]
   (a) Add x to the RD of Ry
   (b) Set AD of x to only Ry
Example [p. 546]

\[ t = a - b \]
\[ u = a - c \]
\[ v = t + u \]
\[ a = d \]
\[ d = v + u \]
\[ t = a - b \]

LD R1, a
LD R2, b
SUB R2, R1, R2
\[
t = a - b
\]

<table>
<thead>
<tr>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>t</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>t</th>
<th>u</th>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

LD R1, a
LD R2, b
SUB R2, R1, R2

No registers are in use - pick the first two available for a and b. Choose to put t in R2 because b is not used again in this block.
\( t = a - b \)

LD R1, a
LD R2, b
SUB R2, R1, R2

\( u = a - c \)

LD R3, c
SUB R1, R1, R3
\[
\begin{align*}
  t &= a - b \\
  u &= a - c
\end{align*}
\]

<table>
<thead>
<tr>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
</tr>
<tr>
<td>a</td>
<td>t</td>
<td>c</td>
</tr>
<tr>
<td>u</td>
<td>t</td>
<td>c</td>
</tr>
</tbody>
</table>

- a is already in R1, so no load needed.
- t is used later, so don't overwrite R2.
- Load c into R3.
- Put result into R1 since a is not needed again in this block.
t = a - b
LD R1, a
LD R2, b
SUB R2, R1, R2

u = a - c
LD R3, c
SUB R1, R1, R3

v = t + u
ADD R3, R2, R1
\[ t = a - b \]

LD R1, a
LD R2, b
SUB R2, R1, R2

\[ u = a - \]

Perform addition, putting the result into R3; c is no longer needed in this block.

ADD R3, R2, R1

\[ v = t + u \]
<table>
<thead>
<tr>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>u</td>
<td>t</td>
<td>v</td>
</tr>
</tbody>
</table>

```
<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>t</th>
<th>u</th>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>R2</td>
<td>R1</td>
<td>R3</td>
</tr>
</tbody>
</table>
```

Same state as at end of previous slide
a = d

LD R2, d
Load d into R2, attach a to R2 as well.
<table>
<thead>
<tr>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>u</td>
<td>t</td>
<td>v</td>
</tr>
</tbody>
</table>

\[ a = d \]

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>t</th>
<th>u</th>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>R2</td>
<td>R1</td>
<td>R3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>u</td>
<td>a,d</td>
<td>v</td>
</tr>
</tbody>
</table>

\[ d = v + u \]

<table>
<thead>
<tr>
<th>R2</th>
<th>b</th>
<th>c</th>
<th>d,R2</th>
</tr>
</thead>
<tbody>
<tr>
<td>R2</td>
<td>b</td>
<td>c</td>
<td>d,R2</td>
</tr>
<tr>
<td>R1</td>
<td>R3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

LD R2, d

ADD R1, R3, R1