Announcements

- Grading survey:
  - Keep current policy. Presentations would take place MWF of the last week, with each of our six teams having ~20 minutes to present their compiler, discuss challenges, and take questions from the class.
  - No presentation, option #1: PR 45%, HW35%, EX 20%.
  - No presentation, option #2: PR 50%, HW30%, EX 20%.
Phases of a compiler

Target machine code generation

Figure 1.6, page 5 of text
"killed" nodes in DAG

When the target of an assignment to an array element, such as \( a[x] = y \), can only be determined at runtime (i.e. when \( x \) is unknown at compile time) we cannot allow subexpression elimination for expressions involving \( a[z] \).
8.5.5 Array References

Array indexing must be handled with care.

"An assignment from an array, like \( x = a[i] \), is represented by creating a node with operator \( =[] \) and two children representing the initial value of the array, \( a_0 \) in this case, and the index \( i \). Variable \( x \) becomes a label of this new node." [p. 537]
\[ x = a[i] \]
"An assignment \textit{to} an array, like \( a[j] = y \), is represented by creating a node with operator \( [ ] = \) and three children representing \( a0, j \) and \( y \). There is no variable labeling this node. What is different is that the creation of this node \textit{kills} all currently constructed nodes whose value depends on \( a0 \). A node that has been killed cannot receive any more labels; that is, it cannot become a common subexpression." [p. 537]
a[j] = y
Ex. 8.13 [p. 538]

1) \( x = a[i] \)
2) \( a[j] = y \)
3) \( z = a[i] \)

Issue: we cannot assume that \( a[i] \) in 3rd statement is the same as \( a[i] \) in the first, as it may be that case that \( i = j \)
Ex. 8.13 [p. 538]

1) \( x = a[i] \)
2) \( a[j] = y \)
3) \( z = a[i] \)
Ex. 8.13 [p. 538]

Because the first node we built depends on $a_0$, it is killed, meaning that no more variables can be added to its label from this point on.

1) $x = a_i$
2) $a_j = y$
3) $z = a_i$
The effect of this is that even though the third statement involves \(a[i]\) we cannot structure share by adding \(z\) as a label next to \(x\). Instead a new node must be constructed, forcing recomputation of this value.
The same issue comes up with pointers, though the effect is more significant.

In general, *p could refer to ANY variable in the program:

\[ x = *p; \]

"As a consequence, the operator =* must take all nodes that are currently associated with identifiers as arguments, which is relevant for dead-code elimination." [p. 539]
Pointers

Similarly, *q could refer to ANY variable in the program:

```
*q = y;
```

"...the *= operator kills all other nodes so far constructed in the DAG." [p. 539]
Function calls

"...we must assume that a procedure uses and changes any data to which it has access. Thus, if procedure \( P \) is in the scope of variable \( x \), a call to \( P \) both uses the node with attached variables \( x \) and kills that node." [p. 539]

Parameter passing plays into this (e.g. pass by value).

In our language we must be careful about arrays and records: like Java, our language passes a pointer (by value).

Is there a concern with functions passed as arguments to functions?
Array Bounds Checking

Remember that we need to do bounds checking of for each array access, e.g. $a[i]$, $m[r,s,t]$. If an array index is out-of-bounds, the program should terminate.
Array Bounds Checking

If for each array access you generate code to extract the size of each dimension of the array:

\[
t_{\text{size}} = \ldots \text{get size}\ldots \\
\text{if } i < t_{\text{size}} \text{ goto AccessOK} \\
\text{goto AccessNotOK}
\]

then during the DAG construction multiple computations to \ldots \text{get size}\ldots will be dropped.
DAG -> three address code

- For each node labelled with variables, generate code to compute the value at the node; assign the value to one of the variables, preferentially one that is live on exit.

- If we do not perform a global liveness analysis, must assume all (non compiler-generated temporary) variables are live on exit.
Example 8.10 [p. 534]

If $b$ is live on exit:

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

If b is not live on exit:

1) \( a = b + c \)
2) \( d = a - d \)
3) \( c = d + c \)
DAG -> three address code
[p. 540 - 541]

1. "The order of instructions must respect the order of nodes in the
   DAG. That is, we cannot compute a node’s value until we have
   computed a values for each of its children."
a = b + c
d = a * b

evaluate b

evaluate c

compute b+c

assign to a

evaluate a

evaluate b

evaluate b

compute a*b

assign to d
a = b + c
d = a * b

evaluate b
evaluate c
compute b+c
assign to a

compute a*b
assign to d

Children are evaluated before parents.
DAG -> three address code
[p. 540 - 541]

1. "The order of instructions must respect the order of nodes in the DAG. That is, we cannot compute a node's value until we have computed a values for each of its children."

This should fall out naturally from the way the value-number algorithm works (see 6.1.2 on page 360).
2. "Assignments to an array must follow all previous assignments to, or evaluations from, the same array, according to the order of these instructions in the original basic block."
2. "Assignments to an array must follow all previous assignments to, or evaluations from, the same array, according to the order of these instructions in the original basic block."

\[
\begin{align*}
a[i] &= x \\
y &= a[j] \\
a[k] &= z
\end{align*}
\]

Cannot put \(a[k] = z\) earlier in sequence
2. “Assignments to an array must follow all previous assignments to, or evaluations from, the same array, according to the order of these instructions in the original basic block.”

\[a[i] = x\]
\[y = a[j]\]
\[a[k] = z\]

Cannot put \(a[k] = z\) earlier in sequence.

This should fall out naturally from the way the value-number algorithm works (see 6.1.2 on page 360).
DAG -> three address code
[p. 540 - 541]

3. "Evaluations of array elements must follow any previous (according to the original block) assignments to the same array. The only permutation allowed is that two evaluations from the same array may be done in either order, as long as neither passes over an assignment to that array."
3. “Evaluations of array elements must follow any previous (according to the original block) assignments to the same array. The only permutation allowed is that two evaluations from the same array may be done in either order, as long as neither passes over an assignment to that array.”

\[ r = a[i] \]
\[ s = a[j] \]
\[ a[k] = z \]
\[ t = a[m] \]

Cannot put \( t = a[m] \) earlier in sequence, but assignments to \( r \) and \( s \) can be reordered.
3. “Evaluations of array elements must follow any previous (according to the original block) assignments to the same array. The only permutation allowed is that two evaluations from the same array may be done in either order, as long as neither passes over an assignment to that array.”

\[
\begin{align*}
  r &= a[i] \\
  s &= a[j] \\
  a[k] &= z \\
  t &= a[m]
\end{align*}
\]

This should fall out naturally from the way the value-number algorithm works (see 6.1.2 on page 360).

Cannot put \( t = a[m] \) earlier in sequence, but assignments to \( r \) and \( s \) can be reordered.
4. "Any use of a variable must follow all previous (according to the original block) procedure calls or indirect assignments through a pointer."
4. "Any use of a variable must follow all previous (according to the original block) procedure calls or indirect assignments through a pointer."

```c
r = a[j];
s = r.m;
f(a, r)
x = a[j];
y = r.m;
```

Since a and r are pointers (passed by value) we must assume that f can change any constituent values of either: don't re-order.
4. "Any use of a variable must follow all previous (according to the original block) procedure calls or indirect assignments through a pointer."

\begin{verbatim}
  r = a[j];
  s = r.m;
  f(a, r)
  x = a[j];
  y = r.m;
\end{verbatim}

Since a and r are pointers (passed by value) we must assume that f can change any constituent values of either: don't re-order.

This should fall out naturally from the way the value-number algorithm works (see 6.1.2 on page 360).
DAG $\rightarrow$ three address code

[p. 540 - 541]

5. "Any procedure call or indirect assignment through a pointer must follow all previous (according to the original block) evaluations of any variable."
5. “Any procedure call or indirect assignment through a pointer must follow all previous (according to the original block) evaluations of any variable.”

\[ x = a[j]; \]
\[ y = r.m; \]
\[ f(a, r) \]

\[ f(a, r) \] may change the values of \( a[j] \) and \( r.m \). Therefore can't put call to \( f \) before those accesses.
5. "Any procedure call or indirect assignment through a pointer must follow all previous (according to the original block) evaluations of any variable."

\[
x = a[j]; \\
y = r.m; \\
f(a,r)
\]

\(f(a,r)\) may change the values of \(a[j]\) and \(r.m\). Therefore can't put call to \(f\) before those accesses.

This should fall out naturally from the way the value-number algorithm works (see 6.1.2 on page 360).
DAG -> three address code
[p. 540 - 541]

"That is, when reordering code, no statement may cross a procedure call or assignment through a pointer, and uses of the same array may cross each other only if both are array accesses, but not assignments to element of the array."
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]
"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]
let's revisit earlier example
(paraphrased from 8.6.2, page 544)

A three-address instruction of the form:

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

1. Use \text{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 \text{LD} \( R_a, a' \) (where \( a' \) is one of the possibly many current locations of \( a \))

3. Similarly for \( b \).

4. Generate \text{OP} \( R_v, R_a, R_b \)
copy instructions

\[ x = y \]

“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 \( \text{LD Ry, y} \). 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 its 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 R1 to only x
   (b) Set AD of x to only R1 (&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 fas for all load statement (per rule 1):" [p. 545]
   (a) Add s to the RD of Ry
   (b) Set AD of x to only Ry