Lesson 14

• More on code generation
Summary

• Next-use information in basic blocks
• A Code Generator
  – Register allocation and assignment
    • Graph coloring
  – Instruction selection
    • Tree transducer
• An overview on Dataflow Analysis and some Global Optimization techniques
A Simple Code Generator

• Algorithm for generating target code for a basic block (sequence of three-address statements) using (local) next-use information

• Critical issue: how to use registers. Several competing uses:
  – To store operands of a target code operation
  – Registers make good temporaries
  – To hold (global) values computed in a block and used in another (e.g., loop indexes)
  – To help runtime storage management (stack pointer, ...)

• The algorithm will check if operands of three-address code are available in registers to avoid unnecessary stores and loads.
(Local) Next-Use Information

• *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 $L: \ x := y \ op \ z$
  – Add liveness/next-use info on $x$, $y$, and $z$ to statement $L$
    • This info can be stored in the symbol table
  – 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 $L$
Next-Use (Step 1)

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

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

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

- Attach current live/next-use information (from symbol table)
- Since the is no info, assume variables are live
- [Data-flow analysis can provide accurate (global) information]
Next-Use (Step 2)

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

\[
\begin{align*}
\text{live}(a) &= \text{true} & \text{nextuse}(a) &= k \\
\text{live}(b) &= \text{true} & \text{nextuse}(b) &= k \\
\text{live}(t) &= \text{false} & \text{nextuse}(t) &= \text{none} \\
\end{align*}
\]

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

- Compute live/next-use information at \( k \) and store in the symbol table

\[ i: \quad b := b + 1 \]
Next-Use (Step 3)

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

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

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

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

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

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

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

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

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

\[ i: \quad b := b + 1 \quad [ \text{live}(b) = \text{true}, \text{nextuse}(b) = j] \]

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

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

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

• We assume that
  – A set of register can be used for values used within the block
  – The order of statements in the block is fixed
  – Each three-address operator corresponds to a single machine instruction
  – Machine instructions take operands in registers and leave the result in a register

• Allowed machine instructions:
  – LD $reg, mem$
  – ST mem, $reg$
  – OP $reg, reg, reg$
Code Generator: sketch

• For each three-address instructions, the algorithm
  – checks which operands are not in register, and emits corresponding loads
  – emits the operation
  – emits the store of the result, if needed

• The algorithm makes use of address descriptors and register descriptors, and of function getreg() such that getreg(x = y OP z) returns the three registers to be used for x, y and z (denoted Rx, Ry and Rz).
Register and Address Descriptors

• For each available register, a register descriptor (RD) keeps track of the vars whose current value is in that register
  – Initially empty
• For each variable, an address descriptor (AD) keeps track of the locations where the current value of the var can be found
  – Location can be a register, a stack location, a memory address, etc.
  – Can be stored in the symbol table
Managing Register and Address Descriptors

• For LD R,x
  – Change RD for R so it holds only x
  – Change AD for x by adding R as an additional location

• For ST x,R
  – Change AD for x to include its own memory location

• For OP Rx,Ry,Rz
  – Change RD for Rx so it holds only x
  – Change AD for x so its only location is Rx
  – Remove Rx from the AD of any variable other than x
The Code Generation Algorithm

• For a three-address instruction, e.g. $x=y \text{ } OP \text{ } z$
  – Use $\text{getReg}(x=y \text{ } OP \text{ } z)$ to select registers $Rx, Ry, Rz$ for $x, y, z$
  – If $y$ is not in $Ry$, emit an instruction $\text{LD} \text{ } Ry, \text{ } y'$ where $y' \subseteq AD(y)$, preferably a register
  – Similarly for $z$
  – Issue the instruction $\text{OP} \text{ } Rx, Ry, Rz$

• Copy statement $x=y$
  – If $y$ is not already in register, emit $\text{LD} \text{ } Ry, \text{ } y'$
  – Adjust RD for $Ry$ so it includes $x$
  – Change AD for $x$ so its only location is $Ry$

• Ending the basic block
  – If $x$ is used at other blocks, issue $\text{ST} \text{ } x, Rx$
Example of code generation (1)

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

- \( t,u \) and \( v \) are temporaries, while \( a, b, c, d \) are global variables
- Assume that registers are enough
  - Reuse registers whenever possible
Example of code generation (2)

\[ t = a - b \]

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

\[ u = a - c \]

LD R3, c; SUB R1, R1, R3
Example of code generation (3)

\[ v = t + u \]

**ADD R3, R2, R1**

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

**LD R2, d**

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

Consider Example of code generation (3)
Example of code generation (4)

\[ d = v + u \]

\[
\begin{array}{ccc}
R1 & R2 & R3 \\
d & a & v \\
\end{array}
\]

ADD R1, R3, R1

\[
\begin{array}{cccccccc}
a & b & c & d & t & u & v \\
R2 & b & c & R1 & & & R3 \\
\end{array}
\]

ST a, R2; ST d, R1

\[
\begin{array}{ccc}
R1 & R2 & R3 \\
d & a & v \\
\end{array}
\]

\[
\begin{array}{cccccccc}
a & b & c & d & t & u & v \\
a,R2 & b & c & d,R1 & & & R3 \\
\end{array}
\]
The *getreg* algorithm

To compute \( \text{getreg}(x := y \text{ OP } z) \)

1. If \( y \) is stored in a register \( R \), return it as \( R_y \)
2. If \( y \) is not in a register, but exists \( R \) empty, return it as \( R_y \)
3. If \( y \) is not in a register and no register is empty, consider \( R \) and check any variable \( v \in RD(R) \)
   a. If \( AD(v) \) does not contain only \( R \), OK.
   b. If \( v = x \) and \( x \) is not an operand in this instruction, OK.
   c. If \( v \) is not used later, then OK.
   d. Otherwise emit \( \text{ST} \ v, R \) this is a *spill*

Choose \( R \) that minimizes the number of *spills* and return it as \( R_y \)

4. Same algorithm for determining \( R_z \)

5. For \( Rx \), similar algorithm, but
   a. Any register containing \( x \) only is OK
   b. It is possible to return \( R_y \) for \( Rx \) if \( y \) is no more used and if \( RD(R_y) = \{y\} \). Similarly for \( R_z \).

To compute \( \text{getreg}(x := y) \)

1. Choose \( R_y \) as above
2. Choose \( Rx = R_y \)
Register Allocation and Assignment

• The code generation algorithm based on `getreg()` is not optimal
  – All live variables in registers are stored (flushed) at the end of a block: this could be not necessary

• *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

• Groups of registers can be dedicated to certain values (base addresses, arithmetics, stack pointer)
  – Simpler code generator, but less efficient use of registers
Allocating Registers in Loops: Usage Counts

• Suppose
  – not storing a variable $x$ has a benefit of 2
  – accessing a variable in register instead of in memory has benefit 1

• Let
  – $use(x, B) =$ number of uses of $x$ in $B$ before assignment
  – $live(x, B) =$ 1 if $x$ is assigned in $B$ and live on exit from $B$

• Then the (approximate) benefit of allocating a register to a variable $x$ within a loop $L$ is

\[
\sum_{B \in L} \left( use(x, B) + 2 \cdot live(x, B) \right)
\]
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): two variables have an edge if one is live where the other is defined.

• Find a $k$-coloring for the graph, with $k$ the number of registers.
Register Allocation with Graph Coloring: Example

```
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

Live range of \( b \)

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

Interference graph

Solve

Three registers:

\[\begin{align*}
a &= r2 \\
b &= r3 \\
c &= r1 \\
d &= r2 \\
e &= r1 \\
f &= r1
\end{align*}\]
On Instruction Selection by Tree Rewriting

• Our simple algorithm uses a trivial Instruction Selection
• In practice it is a difficult problem, mainly for CISC machines with rich addressing mode
• Tree-rewriting rules can be used effectively for specifying the translation from IR to target code
• Tree-translation schemes can be handled with techniques similar to syntax-directed definitions: can be the basis of code-generator generators
• Start with a tree representing the IR code, and reduce it using the rules, producing the target code as associated actions
# Code generation using a Tree Translation Scheme

<table>
<thead>
<tr>
<th>Step</th>
<th>Instruction</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1)</td>
<td>$R_i \leftarrow C_a$</td>
<td>{ LD $R_i$, #a }</td>
</tr>
<tr>
<td>2)</td>
<td>$R_i \leftarrow M_x$</td>
<td>{ LD $R_i$, x }</td>
</tr>
<tr>
<td>3)</td>
<td>$M \leftarrow = \noindent$</td>
<td>{ ST $x$, $R_i$ }</td>
</tr>
<tr>
<td></td>
<td>$M_x$ $R_i$</td>
<td></td>
</tr>
<tr>
<td>4)</td>
<td>$M \leftarrow = \noindent$</td>
<td>{ ST *$R_i$, $R_j$ }</td>
</tr>
<tr>
<td></td>
<td>$R_j$ $R_i$</td>
<td></td>
</tr>
<tr>
<td>5)</td>
<td>$R_i \leftarrow \text{ind}$</td>
<td>{ LD $R_i$, a($R_j$) }</td>
</tr>
<tr>
<td></td>
<td>$C_a$ $R_j$</td>
<td></td>
</tr>
<tr>
<td>6)</td>
<td>$R_i \leftarrow + \noindent$</td>
<td>{ ADD $R_i$, $R_i$, a($R_j$) }</td>
</tr>
<tr>
<td></td>
<td>$R_i$ \text{ind}</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$C_a$ $R_j$</td>
<td></td>
</tr>
<tr>
<td>7)</td>
<td>$R_i \leftarrow + \noindent$</td>
<td>{ ADD $R_i$, $R_i$, $R_j$ }</td>
</tr>
<tr>
<td></td>
<td>$R_i$ $R_j$</td>
<td></td>
</tr>
<tr>
<td>8)</td>
<td>$R_i \leftarrow + \noindent$</td>
<td>{ INC $R_i$ }</td>
</tr>
<tr>
<td></td>
<td>$R_i$ $C_1$</td>
<td></td>
</tr>
</tbody>
</table>

![Figure 8.19: Intermediate-code tree for $a[i] = b + 1$](image)

- $\rightarrow$ LD R0, #a
- $\rightarrow$ ADD R0, R0, SP
- $\rightarrow$ ADD R0, R0, i(SP)
- $\rightarrow$ LD R1, b
- $\rightarrow$ INC R1
- ST *R0, R1
Tree matching

• Various algorithms for pattern matching and general tree matching
• LR parsing on a string (prefix) representation of trees
  – Uses a syntax-directed translation scheme encoding the tree-translation scheme
  – Highly ambiguous. In absence of costs, reduce-reduce conflicts choose the longer reduction, shift-reduce prefer shift
• General tree matching transforming templates into sets of strings, and by matching multiple strings in parallel
• We can associate costs with the tree-rewriting rules and apply dynamic programming to obtain an optimal instruction selection
Optimal Code Generation for Expressions

• We can choose registers optimally
  – If a basic block consists of a single expression, or
  – It is sufficient to generate code for a block one expression at a time
Ershov Numbers

• Assign the nodes of an expression tree a number that tells how many registers needed
  – Label leaf 1
  – The label of an interior node with one child is the label of its child
  – The label of an interior node with two children
    • the larger one if the labels are different
    • One plus the label if the labels are the same
Ershov Numbers

(a-b)+e*(c+d)

\[
\begin{align*}
t1 &= a - b \\
t2 &= c + d \\
t3 &= e \times t2 \\
t4 &= t1 + t3
\end{align*}
\]
Generating Code From Labeled Expression Tree

• Recursive algorithm starting at the root
  – Label k means k registers will be used
  – $R_b, R_{b+1},...,R_{b+k-1}$, where $b>=1$ is a base

• To generate machine code for a node with label k and two children with equal labels
  – Recursively generate code for the right child, using base $b+1$: $R_{b+1}, R_{b+2},...,Result \ in \ R_{b+k-1}$
  – Recursively generate code for the left child, using base $b$: $R_b, R_{b+1},...,Result \ in \ R_{b+k-2}$
  – Generate instruction OP $R_{b+k-1}, R_{b+k-2}, R_{b+k-1}$
Generating Code From Labeled Expression Tree

• To generate machine code for a node with label k and two children with unequal labels
  – Recursively generate code for the child with label k, using base b: $R_b, R_{b+1}, \ldots$, Result in $R_{b+k-1}$
  – Recursively generate code for the child with label m, using base b: $R_b, R_{b+1}, \ldots$, Result in $R_{b+m-1}$
  – Generate instruction OP $R_{b+k-1}, R_{b+m-1}, R_b+1$.
• For a leaf $x$, if base is b generate LD $R_b, x$
Ershov Numbers

LD R3, d
LD R2, c
ADD R3, R2, R3
LD R2, e
MUL R3, R2, R3

LD R2, b
LD R1, a
SUB R2, R1, R2
ADD R3, R2, R3
Insufficient Supply of Registers

• Input: a labeled tree and a number \( r \) of registers
• For a node \( N \) with at least one child labeled \( r \) or greater
  – recursively generate code for the big child with \( b=1 \).
    The result will appear in \( R_r \)
  – Generate machine instruction \( ST \ t_k, R_r \)
  – If the little child has label \( r \) or greater, \( b=1 \). If the label is \( j<r \), then \( b=r-j \). The result in \( R_r \)
  – Generate the instruction \( LD \ R_{r-1}, t_k \)
  – Generate \( OP \ R_r, R_r, R_{r-1} \) or \( OP \ R_r, R_{r-1}, R_r \)
Insufficient Supply of Registers

```
LD R2, d
LD R1, c
ADD R2, R1, R2
LD R1, e
MUL R2, R1, R2
ST t3, R2
```

```
LD R2, b
LD R1, a
SUB R2, R1, R2
LD R1, t3
ADD R2, R2, R1
```