Chapter 7 OCG
I. Introduction
OCG is a phase where the IC is translated into the language of the target machine.
(Assembly)
In this class: IC would be 3AC; OC target machine is IBM 370
There are two types of instructions: R-R-instruction and R-M-instruction.
Example:
\[
\begin{align*}
AR & 2, 3 \quad \text{(R-R instruction)} \\
A & 2, x \quad \text{(R-M instruction)}
\end{align*}
\]

II. Blind Generation
Idea: develop a set of OC instructions per IC instruction.
For example: 3AC: 1. \(a = b + c\)  
\[
\begin{align*}
1. \text{mov ax, } b \\
& \text{add ax, } c \\
& \text{mov a, ax}
\end{align*}
\]
\[
\begin{align*}
2. \text{mov ax, } a \\
& \text{add ax, } b \\
& \text{mov d, ax}
\end{align*}
\]
Disadvantage: redundant/unnecessary instructions.
Solution: Context sensitive generation

III. Context sensitive generation
We have to keep track of three things:
1. keep track of the values in the registers
2. keep track of the values of the variables
3. keep track of: if a variable is going to be used later (3AC instruction) and where.
For 1. We have register association table to store what variable is in each register.

<table>
<thead>
<tr>
<th>R2</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>R3</td>
<td>b</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>R15</td>
<td>...</td>
</tr>
</tbody>
</table>

For 2. We use Address table. Keep track of where (register # or memory) the variables are stored.

<table>
<thead>
<tr>
<th>a</th>
<th>R2</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>R3</td>
</tr>
<tr>
<td>c</td>
<td>M</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
For 3. We use the concept of Liveness & Next-use.

A. Computation of Live and Next-use.
Example:
Source: \( d = (a - b) + (c - a) - (d + b) \times (c + 1) \)

IC (3AC) : 1. \( u = a - b \); 2. \( v = c - a \); 3. \( w = u + v \); 4. \( x = d + b \);
5. \( y = c + 1 \); 6. \( z = x \times y \); 7. \( d = w - z \);

Live and Next-use table:

<table>
<thead>
<tr>
<th></th>
<th>Live</th>
<th>Next-use</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>( u ) (L)</td>
<td>( a ) (L)</td>
</tr>
<tr>
<td>2.</td>
<td>( v ) (L)</td>
<td>( c ) (L)</td>
</tr>
</tbody>
</table>

The computation time would be \( O(n^2) \).

A better computation method (\( O(n) \)) is as the followings:

Step 1: In the symbol table, mark all user defined variables “Live”, and all temporary variables “Dead”.

Step 2: Start from the last instruction, \( \rightarrow \) backwards, for each 3AC instruction \( a = b(op)c \),
do:

a. Look at the symbol table and see whether \( a \), \( b \), \( c \) are live or dead, and the next-user, attach this info to the instruction.
b. update the symbol table. (1) mark the target a “dead” and “no next-use”. (2) For operands b, c, mark them “live” and the next-use is the current instruction.

Example:
Symbol Table:

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>L(2)</td>
<td>L(1)</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>L(4)</td>
<td>L(1)</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>L(5)</td>
<td>L(2)</td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>D</td>
<td>L(4)</td>
<td></td>
</tr>
<tr>
<td>u</td>
<td>D</td>
<td>L(3)</td>
<td></td>
</tr>
<tr>
<td>v</td>
<td>D</td>
<td>L(3)</td>
<td></td>
</tr>
<tr>
<td>w</td>
<td>D</td>
<td>L(7)</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>D</td>
<td>L(6)</td>
<td></td>
</tr>
<tr>
<td>y</td>
<td>D</td>
<td>L(6)</td>
<td></td>
</tr>
<tr>
<td>z</td>
<td>D</td>
<td>L(7)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Liveness</th>
<th>Next-use</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. ( u ) (L)</td>
<td>( a ) (L)</td>
</tr>
<tr>
<td>2. ( v ) (L)</td>
<td>( c ) (L)</td>
</tr>
<tr>
<td>3. ( w ) (L)</td>
<td>( u ) (D)</td>
</tr>
<tr>
<td>4. ( x ) (L)</td>
<td>( d ) (D)</td>
</tr>
<tr>
<td>5. ( y ) (L)</td>
<td>( c ) (L)</td>
</tr>
<tr>
<td>6. ( z ) (L)</td>
<td>( x ) (D)</td>
</tr>
<tr>
<td>7. ( d ) (L)</td>
<td>( w ) (D)</td>
</tr>
</tbody>
</table>

B. Register management:
IBM 370 type of object code:
R-M instruction / R-R instruction.
GetREG(b) function:
for 3AC instruction a = b(op)c
    if (b in register R) and (R does not hold any other variable) and (b is dead) then
        return R;
    else if there is an empty register R
        return R;
    else
        - select an occupied register R to use
        - generate the store instruction to save R’s content
        - update the tables and return R

C. Context sensitive generation of O.C.
Step 1: Ask GetREG(b) for a register R.
   if returns an empty register, then generate:
       L R, b
   if instruction is a = b, then change the descriptor for a to R.
Step 2: Find “c” using the address table.
   if c is in memory, then generate op R, c
   else if c is in register(S), then generate opR R, S
   update the tables.
Step3: If the value of “c” is in a register, and c is dead, then free the register. Update the tables.

IV. Instruction Sequencing (Sethi-Ullman method)
Goal: minimum use of registers.
Idea: We are going to change the sequence of 3AC instructions such that it uses minimum number of registers.
Method:
Step 1. Create AST and label the nodes.
   1. if the operands are leaves, then the left operand has a value of 1, the right has 0.
   2. if the sub-trees have different values (counts), then the count (value) of the root is the greater one.
   3. if the sub-trees have same count k, then the count of root is k+1.
Step 2. Order (re-arrange) 3Acs so that more expensive sub-tree is evaluated first.
   If both sub-trees have same counts, then favor the left (or right) but consistently.
Example: d = (a – b) + (c – a) – (d + b) * (c + 1)
New sequence:
1. T1 = c + 1
2. T2 = d + b
3. T3 = T1 * T2
4. T4 = c – a
5. T5 = T4 – T3
6. T6 = a – b
7. d = T6 + T5
This sequence uses maximum 2 registers.
Suggested homework assignments for chapter 7:
7.2, 7.3, 7.5 a, b, c.