Chapter 7

Object Code Generation

- Statements in 3AC are simple enough that it is usually no great problem to map them to corresponding sequences of machine-language instructions, if some care is taken.
- This is, of course, one of the great attractions of 3AC
- The main issues in Object Code Generation:
  - How to avoid redundant Operations.
  - Which Machine Instructions to use.
  - How to manage Registers.

1. Generating Machine Language from 3AC

- This can be done blindly, instruction by instruction, or it can be done with some thought to the way successive instructions interact and especially to the intelligent use of registers.
  - We will consider the blind method first, since it is the simplest.

- In machines where the number of registers is small or their uses are severely restricted, it may not make a lot of sense to devote a lot of effort to optimizing register use.
- Optimization implies a wide range of options to choose from, and if the optimizations are limited, so is the amount of optimizing we can do.

1.1 Blind Generation

- You must take into account
  - The addressing modes of the operands.
  - result and register restrictions (pairs,…)
- We could write a procedure, one for each type of 3AC, that handles the blind translation.
- These would ignore register allocation, and just do loads to get stuff in, and stores to put stuff back.

1.2 Special Considerations

- The main issues are:
  - taking advantage of special instructions,
  - deciding when to deviate from straight translation.
- The use of special instructions arises because there is often more than one way to do things.
  - this is where peephole optimization is good.
  - procedures for repeated things. (calculating array subscripts)

2. Context-Sensitive Translation and Register Use

- The vast majority of computer instructions use a working register to hold one of its operands.
- Since it takes time to copy from registers to memory, keep things in registers.
- The general rule for register usage is: If a value is in a register, and it is going to be used again soon, keep it in a register.
- The problem is when you run out of registers.

- Therefore, we must keep track of:
  - 1. Which registers are used, and what do they hold.
  - 2. Where the current value of the variable can be found.
  - 3. Which variables will be needed later in the block, and where.
  - 4. Which variables whose current values are in registers must be stored upon exiting the block (these are live variables)

- Since programs may have hundreds of variables (including temporaries created in code generation) this job can easily get out of hand.

2.1 Livens and Next Use

- a variable is live if it is going to be used again in the program.
- programmer defined variables can be assumed live at the end of the block.
- temporaries are assumed to not be alive at the end of a block.

2.2 Descriptor Tables

- **register allocation table** -- current contents of each register (every use of a register updates this table).
- **address table** -- where the current value of each variable may be found (memory, register, both)

2.3 Assigning Registers

- int Get_Register(char *, int & new);
- Pass it the operand
  - 1. Find out if the parameter is already in a register. If so return it.
  - 2. If not, find an empty register and fill it in the register table and set new to true.
  - 3. If there are no empty registers, spill.

2.4 Generating Code

- a := b op c
- 1. R = Get_Register (B, new);
- 2. if(new)
  - L R, B // Load register R with B
- 3. Check address table for C
  - if memory
    - op R, C
  - if register S
    - opR R,S

- Don't forget to free temporaries from registers after they are used.
  - This may require a second function to free the register of some variable.
    free_register (char *name);
- Note: Get_Register may also have to worry about even/odd register pairs in some architectures.

2.5 Instruction Sequencing

- Aho, Sethi, and Ullman [1970] devised a method of re-ordering the instructions to evaluate the basic blocks that require the most registers first.
- This can save on your register use, thereby minimizing spilling of registers.

3. Special Architectural Features

- The code generator should be able to take advantage of any special capabilities provided by the target machine.
  - The PD.-11 provided auto-increment and auto-decrement registers. You could set them to auto-increment before or after they gave you the value.
  - ++i or i++

- There is little to say about exotic instructions, since it depends on what the instruction is, and what you consider exotic.
- One can generally say, the more exotic the instruction, the less use it is likely to be.
- It may be more trouble than it is worth to detect the opportunity to use some unusual instruction unless it saves a truly huge number of conventional instructions.

4. Summary

- Object code generation takes as many forms as there are target machines.
- Some programming languages have been influenced by the instruction sets of the machines on which they were first developed
  - C and i++
  - FORTRAN and if(x) 10, 20, 30
    - reflects the comparison and conditional-jump operations of the original target machine.

- You can learn a great deal by using an interactive debugger to analyze and study the object code generated by other compilers for the same machine.
  - You may spot some things the compiler does that are stupid.
  - It may also alert you to problems you can avoid.